МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ОДЕСЬКИЙ ДЕРЖАВНИЙ ЕКОЛОГІЧНИЙ УНІВЕРСИТЕТ

В.Б. Рольщиков

ЗАСТОСУВАННЯ ЗАСОБІВ ІНТЕРФЕЙСУ ПЕРЕДАЧІ ПОВІДОМЛЕНЬ ПРИ ПРОГРАМУВАННІ РОЗПОДІЛЕНИХ СИСТЕМ МОВОЮ JAVA

Навчальний посібник

Одеса ОДЕКУ – 2017

ББК 32.973.2-018

Р68

УДК 37.048.43:004.428.4

Друкується за рішенням. Вченої ради Одеського державного екологічного університету (протокол № _____ від «____» _____ 201 р.

Рольщиков В.Б.

Р68 Застосування засобів інтерфейсу передачі повідомлень при програмуванні розподілених систем мовою Java: Навчальний посібник – Одеса: ОДЕКУ 2017. – 209 с.

Даний навчальний посібник містить опис засобів розробки паралельних програм мовою Java для розподілених обчислювальних комплексів на основі бібліотеки класів MPJ Express – Java-реалізації стандарту MPI (Message Passing Interface). Разом з розглядом основних можливостей бібліотеки посібник містить приклади паралельних програм, які ілюструють основні принципи й прийоми паралельного програмування в середовищі MPI. Для полегшення трудомісткого процесу пошуку помилок програмування наведені дані про застосування системи програмування Eclipse.

Посібник призначений для використання при виконанні лабораторних робіт з дисципліни «Технології розподілених систем та паралельних обчислень» при підготовці студентів по проблематиці паралельних обчислень. Також посібник може бути корисним всім фахівцям широкого профілю, які бажають ознайомитись з бібліотекою класів для програмування у стандарті MPI мовою Java для вирішення складних наукових і прикладних завдань.

© Одеський державний екологічний університет, 2017

ЗМІСТ

Вступ 6
1 Встановлення, структура і настроювання програмного забезпечення MPI для програмування мовою Java 11
1.1 Встановлення і структура MPJ Express 12
1.2 Настроювання MPJ Express у багатоядерній конфігурації 16
1.2.1 Настроювання у системах відкритого типа 16
1.2.2 Настроювання у системах сімейства Windows 17
1.3 Настроювання MPJ Express у конфігурації для різних кластерів 20
1.3.1 Кластерна конфігурація з драйвером niodev 22
1.3.2 Конфігурування кластера з драйвером mxdev 24
1.3.3 Настроювання MPJ Express в конфігурації з драйвером hybdev 25
1.3.4 Конфігурування кластера з драйвером native, з використанням бібліотеки MPI 26
1.4 Компіляція і виконання паралельних MPJ Express програм 28
1.5 Контрольні запитання та завдання для самостійної роботи 31
2 Інтегроване середовище розробки Eclipse як інструмент для написання й налагодження MPJ Express програм 32
2.1 Створення програми у середовищі Eclips 33
2.2 Запуск паралельних програм у Eclips 35
2.3 Налагодження програм за допомогою вбудованого налагоджувача 39
2.4 Робота в Eclipse з зовнішнім налагоджувачем системи MPJ Express 41
2.5 Встановлення плагіну MPJ Express Debugger в інтегрованому середовищі Eclipse 42
2.6 Створення, конфігурування і запуск MPJ Express застосування в багатоядерній конфігурації 43
2.7 Запуск MPJ Express застосування у режимі кластера 45
2.8 Налагодження MPJ Express застосування за допомогою MPJ Express Debugger 46
2.8.1 Процес локального налагодження 46

2.8.2 Віддалене налагоджування 49
3 Склад пакету, ієрархія класів та їхні методи 52
3.1 Склад пакету і дерево ієрархії класів 52
3.2 Розподіл методів класів по функціональнім групам пакету MPJ Express 56
3.3 Контрольні запитання та завдання для самостійної роботи 60
4 Застосування методів пакету MPJ Express для створення паралельних програм 60
4.1 Поняття груп, комунікаторів, ідентифікаторів процесів і повідомлень, механізми обробки помилок у пакеті MPJ Express 61
4.1.1 Контрольні запитання та завдання для самостійної роботи 64
4.2 Методи MPI Express загального призначення 65
4.2.1 Контрольні запитання та завдання для самостійної роботи 72
4.3 Операції комунікації типа «точка-точка» між окремими процесами 73
4.3.1 Передача/прийом повідомлень з блокуванням 75
4.3.2 Передача/прийом повідомлень без блокування 93
4.3.3 Відкладені запити на взаємодію 105
4.3.4 Виникнення тупикових ситуацій (deadlock) і способи їхнього усунення 109
4.3.5 Контрольні запитання та завдання для самостійної роботи 114
4.4 Засоби колективної комунікації процесів 117
4.4.1 Контрольні запитання та завдання для самостійної роботи 152
4.5 Операції з групами процесів і комунікаторами 154
4.5.1 Операції із групами процесів 155
4.5.2 Операції з комунікаторами 163
4.5.3 Контрольні запитання та завдання для самостійної роботи 167
4.6 Створення і робота з віртуальними топологіями процесів 168
4.6.1 Створення декартової топології і робота з нею 169
4.6.2 Створення і робота з віртуальною топологією графа 176
4.6.3 Контрольні запитання та завдання для самостійної роботи 181

4.7 Засоби пересилання різнотипних даних 183
4.7.1 Похідні типи даних 184
4.7.2 Упакування даних 194
4.7.3 Контрольні запитання та завдання для самостійної роботи 197
Ліцензія на пакет програмного забезпечення MPJ Express версії 0.44 199
Перелік посилань 200
Алфавітний покажчик по методах, константах та деяких класах пакета MPJ 202
Перелік рисунків 206
Перелік таблиць 209

ВСТУП

У сьогодення неможливо уявити собі ефективну організацію роботи без застосування комп'ютерів у таких галузях, як планування й управління виробництвом, проектування й розробка складних технічних пристроїв, видавнича діяльність − словом, у всіх галузях, де виникає необхідність в обробці великих обсягів інформації і виконання величезних обсягів обчислень.

Коло завдань з атомної енергетики, авіабудування, ракетно-космічних технологій, моделювання кліматичних процесів і великої низки інших галузей науки й техніки потребують для свого вирішення застосування потужних обчислювальних ресурсів, і таке коло постійно розширюється. Внаслідок широкого впровадження обчислювальної техніки значно розвинувся напрямок чисельного моделювання й чисельного експерименту. Чисельне моделювання, заповнюючи проміжок між фізичними експериментами й аналітичними підходами, дозволило вивчати явища, які є або занадто складними для дослідження аналітичними методами, або такими, що потребують занадто багато коштів, або є небезпечними для експериментального вивчення. При цьому чисельний експеримент дозволив значно здешевити процес наукового й технологічного пошуку. Стало можливим моделювати в реальному часі процеси інтенсивних фізико-хімічних й ядерних реакцій, глобальні атмосферні процеси, процеси економічного й промислового розвитку регіонів і т.ін.

В наслідок цілої низки причин, з існуючих на цей час 4-х класів обчислювальних систем в Україні значне поширення дістали тільки персональні комп'ютери, і відповідно ситуація з освоєнням технологій роботи із цими комп'ютерами більш-менш благополучна. Але для розв'язання вказаних вище завдань можливостей персональних комп'ютерів виявляється дуже замало не зважаючи на постійне збільшення їхньої потужності. Тому виникає потреба у використанні багатопроцесорних комплексів паралельної дії.

Багатопроцесорні системи, які раніше здебільшого асоціювалися із суперкомп'ютерами, у цей час міцно затвердилися у всьому діапазоні існуючих обчи-

слювальних систем, починаючи від персональних комп'ютерів і закінчуючи суперкомп'ютерами на базі векторно-конвеєрних процесорів. Найбільшою популярністю користуються мультікомп'ютери, тобто кластерні багатопроцесорні системи, як централізовані, так і децентралізовані кластери робочих станцій, а також дуже схожі на них масивно паралельні процесори. Таки системи відносно дешеві (особливо кластери робочих станцій) і прості у експлуатації. Ці обставини з одного боку збільшують доступність суперкомп'ютерних технологій, а з іншого, підвищують актуальність їхнього освоєння, оскільки для таких типів багатопроцесорних систем потрібно використання спеціальних технологій програмування для того, щоб програма могла повною мірою використовувати ресурси високопродуктивної обчислювальної системи.

У конспекті лекцій з курсу «Технології розподілених систем та паралельного програмування» [1, 2] докладно розглянуті різні механізми поділу програм на паралельні гілки кожна з яких може виконуватися на окремому процесорі незалежно від інших, а також системи високопродуктивних комп'ютерів і структура програмного забезпечення для паралельної роботи таких систем. Як відомо мультиком'ютерні кластерні системи не мають загальної пам'яті, яка спільно розділяється всіма процесорами, навпаки ці системи мають розподілену пам'ять частини якої належать кожному окремому процесору.

Основним способом взаємодії паралельних процесів у таких системах є передача повідомлень від одного процесора до іншого через спеціальну комунікаційну мережу. На сьогодні однією з найпоширеніших технологій програмування для паралельних комп'ютерів з розподіленою пам'яттю є технологія MPI [3, 4]. Це й відбито в назві даної технології − Message Passing Interface (інтерфейс пересилання повідомлень).

Комунікаційна бібліотека MPI є загальновизнаним стандартом у паралельному програмуванні з використанням механізму передачі повідомлень. MPI-програма являє собою набір незалежних процесів, кожний з яких виконує якунебудь окрему частину єдиної програми, що написана високорівневою мовою програмування.

Як відомо [2] існують дві парадигми створення паралельних програм. По-перше це технологія розпаралелювання даних, коли у багатопроцесорній системі кожен з процесорів виконує ті ж самі обчислення, що і інші, але над власним набором даних. По-друге – парадигма розпаралелювання задач, при якої всі процесори виконують власні незалежні від інших частини однієї програми у загальному випадку оперуючи власними наборами даних. Бібліотеки MPI дозволяють створювати програми з використанням, як першої, так і другої парадигми програмування.

Розробка цього програмного продукту почалась ще у 1994 році з появи так званого Message-Passing Interface Forum (MPIF) [5] за участю більш ніж 40 різних IT організацій. MPIF обсудив і сформулював основні вимоги до розроблювальної системи. Однією з таких вимог була вимога того, щоб розроблювана система обов'язково підтримувала можливість створення паралельних програм за обома технологіями. За результатами роботи форуму був виданий документ «A Message-Passing Interface Standard Version 1.0» – стандарт у якому встановлювався інтерфейс, що його повинні дотримуватися, як відповідні системи програмування на кожній обчислювальній платформі, так і користувачі при створенні своїх власних паралельних програм.

Трохи пізніше у 1995 році з'явився стандарт MPI версії 1.1 [6], який вже остаточно визначав базовий набір функцій необхідний для високорівневого створення системи паралельних процесів, керування ними і здійснення їхньої взаємодії у багатопроцесорній системі з розподіленою пам'яттю. Абсолютно всі сучасні реалізації API MPI у повному обсязі підтримують вимоги цього стандарту і надають можливість створювати доволі складні паралельні програми, які ефективно працюють на всіх системах з розподіленою пам'яттю і, навіть, у системах з поділюваною пам'яттю.

Розвиток і ускладнення паралельних обчислювальних систем постійно вимагали відповідного ускладнення систем програмування. Так у 1997 році MPIF був виданий стандарт MPI-2.0, який значно розширив функціональність попередньої версії. В цьому стандарті були розширені засоби колективної кому-

нікації, додана можливість динамічного створення процесів і паралельного введення/виведення даних, зроблена ще низка доповнень, які не мають суттєвого впливу на якість створюваних паралельних програм.

Пізніше були розроблені і видані ще декілька стандартів (версія 1.3 у 2008 році, MPI-2.1 у 2008 році і у 2009 році MPI версії 2.2) однак ці мінорні версії стандартів не торкалися фундаментальних основ технології MPI, а лише робили деякі «косметичні» зміни стандарту 2.0 і виправляли виявлені помилки і неточності попередніх версій.

Нарешті у 2012 році вийшов у світ MPI стандарт версії 3.0. На відміну від MPI-2.2, цей стандарт вважається серйозним оновленням стандарту MPI. Стандарт MPI-3 містить значні розширення функціональних можливостей API MPI, у частковості додані функції, які не блокують виконання колективних операцій, функції нових односторонніх операцій зв'язку, зроблена прив'язка до системи програмування Fortran 2008. Нові можливості введені у зв'язку з появою переконливих потреб користувачів. Однак, деякі додані функції мають незначний вплив на існуючі реалізації MPI, наприклад, поява функцій створення і керування віконним інтерфейсом програми не є конче потрібною тому, що практично всі сучасні мови програмування самі с успіхом підтримують такі властивості.

На сьогодні вже затверджена і діє реалізація стандарту MPI версії 3.1, яка введена у 2015 році [5].

Історично склалося так, що практична реалізація технології MPI для різних мов програмування спочатку була реалізована як бібліотека функцій для застосування в програмах мовою Fortran 77 (Fortran 90). Така система програмування була вибрана тому, що мова Fortran у той час все ще залишалася (та й досі залишається) доволі розповсюдженою мовою прикладного програмування для створення різних систем моделювання у галузі фізичних і інших природничих наук. В той же самий час велику популярність вже дістала мова програмування C, тому практично одразу була створена і бібліотека функцій для програмування цією мовою. Поступове впровадження технології MPI у практику паралельного

програмування змушувало розробників програмного забезпечення для паралельних систем створювати бібліотеки, які задовольняють стандарту MPI і для інших мов програмування. Так була створена бібліотека класів для мови C++, останнім часом створені реалізації стандарту MPI майже для всіх популярних на сьогодні мов програмування, в тому числі є декілька бібліотек класів для застосування у паралельних програмах мовою Java.

Не дивлячись на широке розповсюдження технології MPI і впровадження відповідних стандартів у різні SDK, на сьогодні склалася доволі дивна ситуація, яка полягає в тому, що практично (якщо б не сказати повністю) відсутня технічна документація і навчальна література про особливості застосування бібліотек MPI в програмах різними мовами програмування окрім мов Fortran і C. Навіть на офіційній сторінці розробників одного з API для мови Java [7] така інформація надана у доволі стислому вигляді і стосується здебільшого питань структури програмного забезпечення і процесу його встановлення на різні операційні системи. Заради справедливості слід сказати, що для мов програмування Fortran і C все ж таки існує доволі багато навчальної літератури англійською (навіть стандартний документ MPI [5] визначає прив'язку лише для цих мов програмування) і російською, наприклад, [8, 9]. А якщо казати про літературу і документацію українською в тому числі і для програмування мовою Java, то її немає взагалі.

Останнім часом популярність Java призвела до того, що мова серйозно розглядається як добра альтернатива мовам Fortran і C і, навіть, C++ при розробці наукових і інженерних застосувань, а також, зокрема, для паралельних обчислень [10].

Вказані обставини і змусили написати навчальний посібник по-перше українською і по-друге такий, що пояснює і надає приклади використання засобів стандарту MPI при програмуванні мовою Java. Окрім того даний навчальний посібник є спробою надати студентам допомогу при виконанні лабораторних робот мовою Java з використанням методів класів пакету MPJ Express під час вивчення дисципліни «Технології розподілених систем та паралельних обчис-

лень». Метою посібника є надання початкових знань в галузі програмного забезпечення паралельних обчислювальних систем і навчання студентів створювати паралельні програми із застосуванням технології MPI.

2 ВСТАНОВЛЕННЯ, СТРУКТУРА І НАСТРОЮВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ MPI ДЛЯ ПРОГРАМУВАННЯ МОВОЮ JAVA

Як вже вказувалося вище, для програмування мовою Java існує декілька пакетів класів, які реалізують інтерфейс MPI. У зв'язку з цим, ще наприкінці 1998 року для Java прив'язки стандарту передачі повідомлень була сформована робоча група «Message-Passing Working Group of the Java Grande Forum». Ця група виступила з початковим проектом для загальної специфікації API. Пізніше у проект були внесені деякі зміни і поточний API для MPI тепер відомий як MPJ і поступово стає еталонним інтерфейсом при програмуванні мовою Java.

Завдяки внесеним змінам MPJ API дещо відрізняється від первісної реалізації, яка відповідає специфікації mpiJava API 1.2. Різниця між цими двома API, полягає в основному в схемах іменування класів і методів. Функціональні можливості, що надаються як API, для користувачів, по суті, ті ж самі.

При написанні навчального посібника, у якості програмного забезпечення, яке підлягає розгляданню, був вибраний пакет класів специфікації MPJ API, який носить назву MPJ Express (MPJE) і на момент написання посібника на офіційному сайті розробника програмного забезпечення – http://mpjexpress.org/ був доступний як MPJ Express Release 0.44 від квітня 2015 року.

Пакет поставляється за дозвільною ліцензією MIT (Massachusetts Institute of Technology). На Російській сторінці Вікіпедії [11] ліцензія MIT описується як GPL-сумісна ліцензія вільного програмного забезпечення, тобто така, що дозволяє програмістам комбінувати і розповсюджувати GPL продукти з ПЗ, яке використовує MIT-ліцензію. ПЗ і супутню документацію під цією ліцензією можна використовувати безоплатно без будь-яких обмежень. Навіть надається необмежене право на використання, копіювання, зміну, публікацію, розповсюдження, субліцензування та/або продаж копій ПЗ. Накладається лише одна вимога: по-

відомлення про авторське право і про умови використання ПЗ і документації обов'язково повинні бути включеними до всіх копій ПЗ або до значимих його частин, що і зроблено у даному посібнику (див. додаток 0).

До недоліків обраного пакету можна віднести підтримання здебільшого лише функцій стандарту MPI - 1.1. Однак, слід сказати, що й дотепер навіть варіант стандарту 2.0 MPI у повному обсязі не одержав широкого втілення у багатьох реалізаціях MPI API. Майже повністю його підтримку здійснюють лише коштовні комерційні проекти. Зі свобідного ПЗ на сьогодні можна відзначити хиба що пакет функцій MPICH, розробники якого анонсують підтримку стандарту 2.0. А підтримка останньої версії стандарту 3.1 поки що тільки почалася.

Але цей недолік зовсім не заважає знайомитись з принципами технології MPI і створювати цілком працездатні і доволі складні програми паралельних обчислень.

2.1 Встановлення і структура MPJ Express

Пакет класів MPJ Express, як практично все ПЗ для Java, є вільно розповсюджуваним, тому його останню версію можна завантажити зі сторінки розробника за адресою https://sourceforge.net/projects/mpjexpress/files/releases/. Отриманий архів просто розпаковується у будь-яке придатне місце у файловій системі, наприклад, для Windows це може бути «D:\SDK\Java» і «/usr/share» для Linux. На рис. 2.1 наведена структура каталогів розпакованого ПЗ.

Безпосередньо у каталозі mpj-v0_44 містяться два текстових файли – README для користувачів Linux і README-win для користувачів Windows в яких наводяться відомості про настроювання пакет класів, компіляцію і запуск розроблених Java-програм у відповідних ОС при багатоядерній або кластерній конфігурації обчислювальної системи. Каталог mpj-v0_44/bin призначений для зберігання скриптів за допомогою яких виконується запуск відкомпільованих Java-програм у тій чи інший ОС. Каталоги mpj-v0_44/src і mpj-v0_44/lib містять відповідно вихідні коди і байт-коди класів пакету mpj. Уміст доволі розвинутого каталогу mpj-v0_44/test призначений для тестування працеспроможності всіх

методів класів пакету, а також може розглядатись як набір прикладів написання програм за допомогою пакету MPJ Express.

```
mpj-v0_44
├──bin
├──conf
├──debugger
│   ├──.settings
│   ├──bin
│   ├──icons
│   │   └──back
│   ├──META-INF
│   └──src
│       └──mpj_express_debugger
├──doc
│   ├──images
│   └──javadocs
│       ├──mpi
│       └──resources
├──lib
├──logs
├──src
│   ├──mpi
│   ├──mpjbuf
│   ├──mpjdev
│   │   ├──javampjdev
│   │   └──natmpjdev
│   │       └──lib
│   ├──runtime
│   │   ├──common
│   │   ├──daemon
│   │   ├──daemonmanager
│   │   └──starter
│   └──xdev
│       ├──hybdev
│       ├──mxdev
│       │   └──lib
│       ├──niodev
│       └──smpdev
└──test
    ├──jgf_mpj_benchmarks
    │   ├──jgfutil
    │   ├──section1
    │   ├──section2
    │   │   ├──crypt
    │   │   ├──lufact
    │   │   ├──series
    │   │   ├──sor
    │   │   └──sparsematmult
    │   └──section3
    │       ├──moldyn
    │       ├──montecarlo
    │       └──raytracer
    ├──microbenchmarkmpiJava
    │   ├──allgather
    │   ├──allreduce
    │   ├──alltoall
    │   ├──barrier
    │   ├──broadcast
    │   ├──gather
    │   ├──pingpong
    │   │   ├──Bsend
    │   │   ├──Rsend
    │   │   ├──Send
    │   │   └──Ssend
    │   ├──reduce
    │   ├──reducescatter
    │   ├──scan
    │   └──scatter
    ├──mpi
    │   ├──ccl
    │   ├──ccl_ObjSer
    │   │   ├──allreduce0
    │   │   ├──reduce0
    │   │   ├──reduce_scatter0
    │   │   └──scan0
    │   ├──comm
    │   ├──dtyp
    │   ├──dtyp_ObjSer
    │   ├──env
    │   ├──group
    │   ├──mpi
    │   ├──perf
    │   ├──pt2pt
    │   ├──pt2pt_ObjSer
    │   ├──signals
    │   ├──threads
    │   └──topo
    ├──mpjdev
    │   ├──buffertest
    │   ├──commtest
    │   ├──grouptest
    │   ├──misc
    │   ├──nbcomms
    │   └──perftest
    ├──nativetest
    └──xdev
```

Рисунок 2.1 − Структура каталогів пакета класів MPJ Express Version 0.44

Особливо слід відзначити каталог mpj-v0_44/doc у якому в форматах .txt,

.doc, .docx та .pdf зібрана деяка документація (англійською) у якій стисло описується застосування багатопоточного налагоджувача і інтегрованого середовища розробки Eclips. Крім того у каталозі mpj-v0_44/doc/javadoc міститься файл index.html інтерактивної веб-сторінки (рис. 2.2) допомоги з класів пакету та всіх їхніх методів.

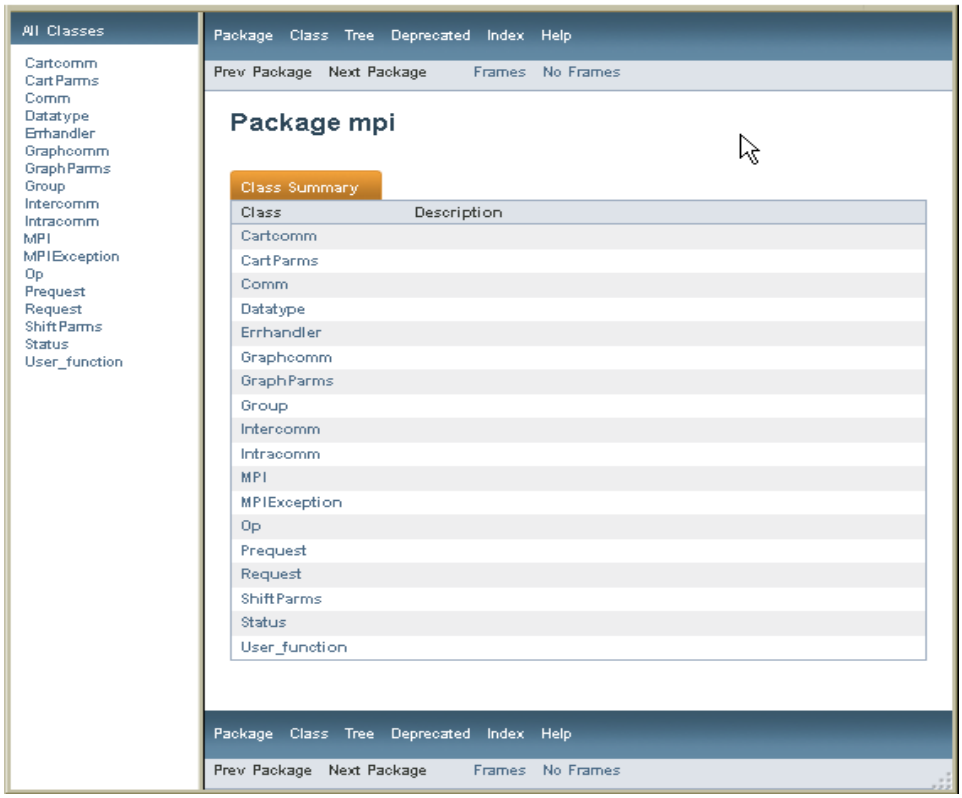


Рисунок 2.2 – Зовнішній вигляд початкової сторінки інтерактивної допомоги пакету MPJ Express

Після розпакування архіву слід провести настроювання пакету для роботи у відповідній операційній системі. Згідно з даними розробників пакету класів MPJ Express [12], програми, які створені з використанням методів класів можуть працювати у двох основних режимах: багатоядерному на процесорах з декількома ядрами і у багатопроцесорному на кластері робочих станцій. Можливим є також використання комбінованого режиму виконання програм (рис. 2.3). При цьому передбачається, що в будь-якому режимі на всіх машинах встановлена Java однієї й тієї ж версії і не менш ніж 1.6.

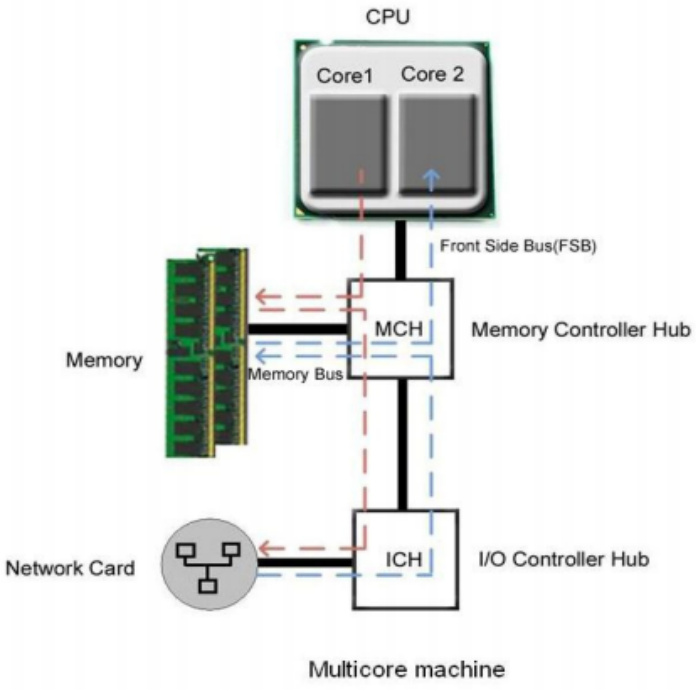


Рисунок 2.3 − Комунікації з використанням сокетів на багатоядерній машині або машині з загальною пам’яттю [12]

Крім того на машинах бажано мати ще деяке додаткове ПЗ.

По-перше це Apache ant 1.6.2 або вищої версії, який може знадобитися для компіляції вихідного коду MPJ Express (наприклад з каталогу mpj-v0_44/test).

По-друге інтерпретатор Perl. Його наявність може знадобитися тому, що деякий Java код генерується за допомогою шаблонів Perl. Спеціальний файл build.xml (розташований у каталозі mpj-v0_44) створює Java код з шаблонів Perl, якщо він виявляє наявність Perl на машині.

Ну і нарешті, бажано мати так звану *споріднену* MPI *бібліотеку*, наприклад, MPICH, Open MPI (для Linux машин) або MS-MPI (для Windows машин), яка може знадобитися при запуску програми з MPJ Express на кластері зі спорідненими пристроями.

Додатково для Linux машин рекомендується наявність утиліти CMake, яка потрібна для генерації makefile, який, в свою чергу, потрібний для створення JNI бібліотеки для пристроїв зі спорідненими бібліотеками.

На рис. 2.4 показані всі можливі конфігурації обладнання, які можуть бу-

ти забезпечені у програмах на основі MPJ Express. Найбільш простою є багатоядерна (комп'ютер з поділюваною пам'яттю) конфігурація обчислюваної системи, відповідно і настроювання пакету для такої системи є найбільш простим.

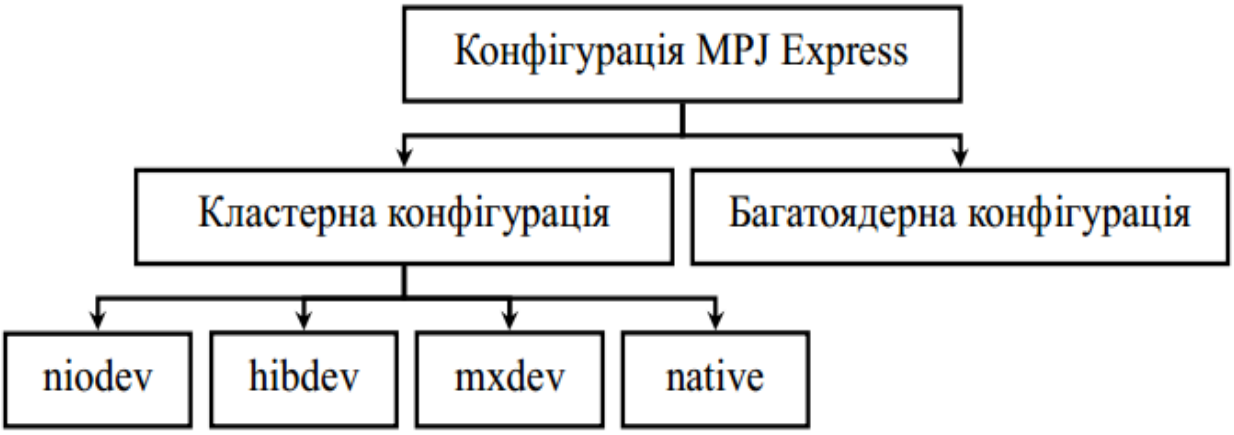


Рисунок 2.4 − Можливі конфігурації запуску програм на основі MPJ Express

2.2 Настроювання MPJ Express у багатоядерній конфігурації

Настроювання пакету MPJ Express для різних операційних систем у багатоядерній конфігурації виконується практично однаково шляхом встановлення значень деяких змінних оточення з урахуванням особливостей відповідної ОС.

2.2.1 Настроювання у системах відкритого типа

Настроювання у багатоядерному режимі для систем відкритого типа UNIX, Linux, MacOS (надалі просто Linux) виконується шляхом привласнення значень змінним оточення операційної системи. У першу чергу встановлюється змінна з ім'ям `MPJ_HOME`. Значення цієї змінної вказує операційній системі де розташовані і де ОС може знайти необхідні їй файли пакету. Робиться це за допомогою наступної системної команди, яка просто подається з консолі Linux:

```
sudo export MPJ_HOME=/path/to/mpj.
```

Замість абстрактного шляху до файлів пакету `/path/to/mpj` слід задавати

дійсний повний шлях до файлів пакету, починаючи з кореневого каталогу, тобто для прикладу розпакування пакету, який наведений вище (див. стор 12) цей шлях буде виглядати таким чином `/usr/share/mpj-v0_44`.

Потім на основі встановленої змінної `MPJ_HOME` виконується зміна значення змінної оточення `PATH` наступним чином:

```
sudo export PATH=$PATH:$MPJ_HOME/bin.
```

Щоб при кожному сеансі роботи з MPJ Express не виконувати дії по встановленню змінних оточення, ці дві команди ОС необхідно занести до вмісту файла `.bashrc`. Слід оговорити, що команди вміщуються в цей файл тільки в разі коли у якості командної оболонки системи застосовується `bash`, у противному випадку команди заносяться до файла, який відповідає обраній оболонці. Після внесення змін у файл `.bashrc` треба командою

```
sudo source .bashrc
```

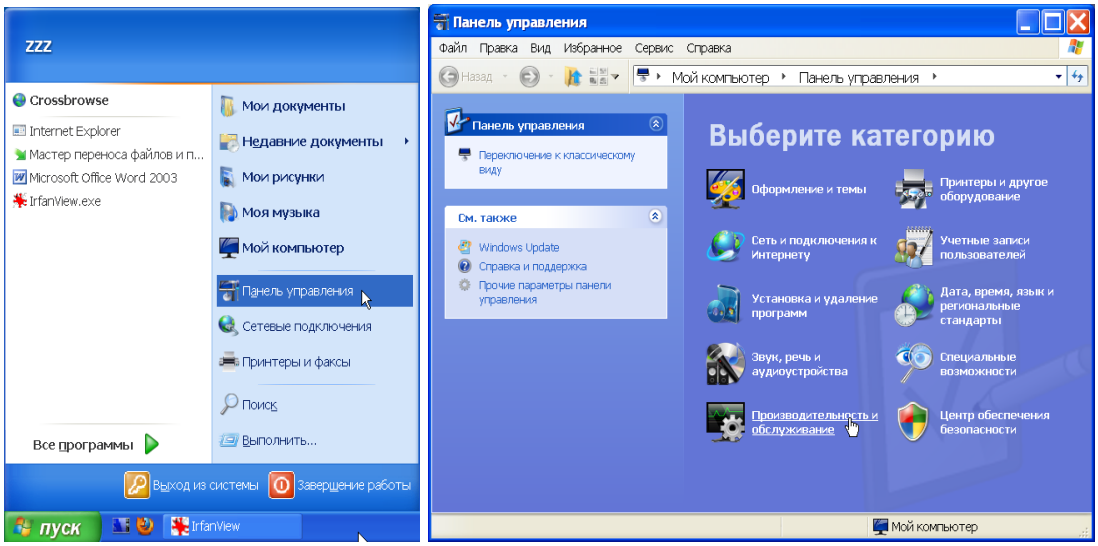
провести ініціювання зроблених змін і повідомити системі про нові значення змінних.

2.2.2 Настроювання у системах сімейства Windows

У системах сімейства Windows встановлення змінних оточення частіше за все виконується за допомогою звичного віконного інтерфейсу, наприклад, наступним чином. В меню «Пуск» системи вибрати пункт «Панель управления» (рис 2.5 а). У вікні «Панель управления», що з'являється після цього, клацнути лівою кнопкою миші на пункті «Производительность и обслуживание» (рис 2.5 б).

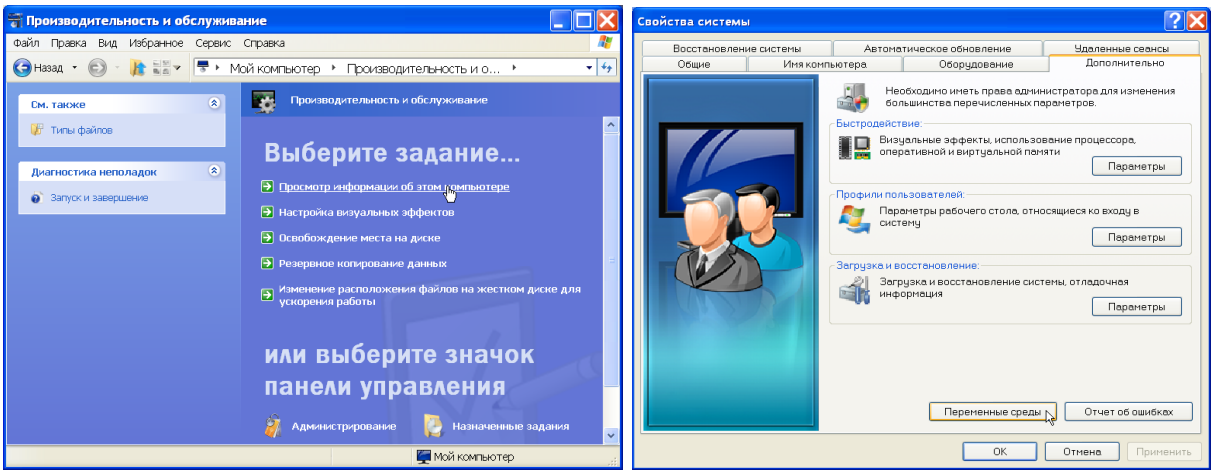
У знову відчиненому вікні «Производительность и обслуживание» (рис. 2.6 а) вибрати завдання «Просмотр информации об этом компьютере», після чого з'являється вікно «Свойства системы» з багатьма вкладками. З цих вкладок необхідно вибрати вкладку, яка має ім'я «Дополнительно» (рис. 2.6 б) і на ній

нажати кнопку «Переменные среды».



а) б)

Рисунок 2.5 – Початок процедури встановлення значень змінних оточення ОС а) – вибір пункту «Панель управления» в основному меню системи; б) – зовнішній вигляд вікна «Панель управления» і вибір категорії «Производительность и обслуживание»

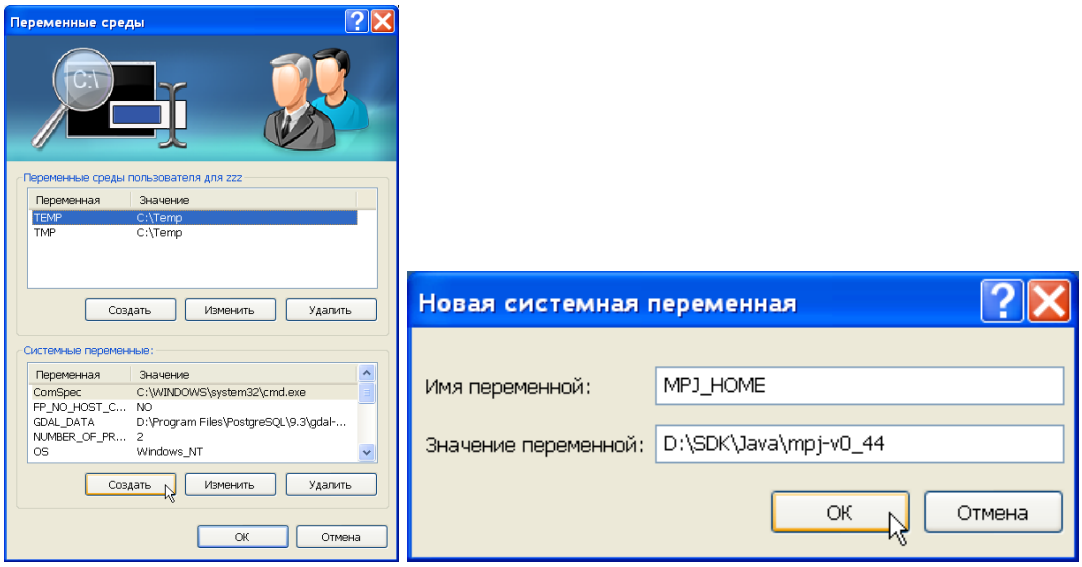


а) б)

Рисунок 2.6 – Зовнішній вигляд вікон «Производительность и обслуживание» з вибором завдання «Просмотр информации об этом компьютере» – а) і «Свойства системы» з вибраною вкладкою «Дополнительно» – б)

Ну і нарешті, у новому діалоговому вікні з ім’ям «Переменные среды»

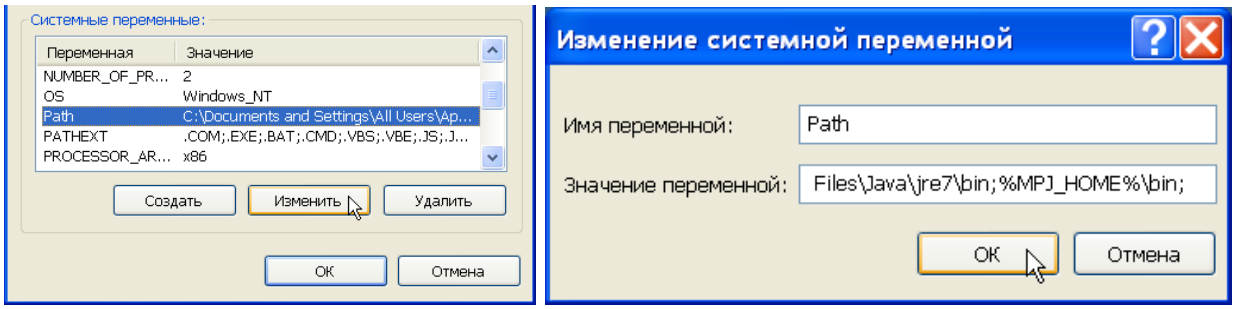
(рис. 2.7 а), обов’язково у розділі «Системные переменные», надавити кнопку «Создать». На останньому етапі, у рядку введення «Имя переменной» діалогового вікна «Новая системная переменная» ввести `MPJ_HOME` – ім’я нової змінної (рис. б). У другому рядку введення – «Значение переменной» того ж самого вікна вводиться повний шлях до каталогу в якому встановлено програмне забезпечення пакету. Для прикладу на стор. 12 повний шлях визначається як `D:\SDK\Java\mpj-v0_44`.



а) б)

Рисунок 2.7 – Загальний вигляд вікна «Переменные среды» з кнопкою «Создать» в блоці «Системные переменные» – а) і вікно «Новая системная переменная» з введеними ім’ям і значенням змінної MPJ_HOME – б)

Після введення значення нової змінної `MPJ_HOME` знов слід повернутися до діалогового вікна «Переменные среды» і його розділу «Системные переменные» і в переліку існуючих змінних оточення вибрати рядок зі змінною на ім’я `PATH`, а потім натиснути кнопку «Изменить», або двічі клацнути лівою кнопкою миші на цьому рядку (рис 2.8 а). На завершення процесу у вікні «Новая системная переменная» доповнити існуюче значення змінної наступним рядком `;%MPJ_HOME%\bin;` не забувши обов’язково почати його символом точки з комою і заключити ім’я змінної `MPJ_HOME` символами відсотків з обох сторін.



а) б)

Рисунок 2.8 – Демонстрація процесу змінення змінної PATH

а) – вибір існуючого значення змінної у вікні «Переменные среды»;

б) доповнення значення новим шляхом «;%MPJ_HOME%\bin;»

Є цілком зрозумілим, що всі ці кроки слід виконувати від ім’я адміністратора системи, у звичайних користувачів прав на зміну значень системних змінних оточення немає.

Для операційної системи Windows 10 настройки значень системних змінних оточення можна здійснити виконавши декілька інший ланцюжок кроків. Клацання правою кнопкою миші на іконці «Этот компьютер» на робочому столі, потім пункт «Свойства» у контекстному меню, далі → «Дополнительные параметры системы» → «Дополнительно» → «Переменные среды». Після чого слідує повторення дій, які були описані вище (див. пояснення до рис. 2.7, 2.8).

Цього ж результату можна досягти скориставшись інструментом «Поиск в Windows» (стилізоване зображення лупи праворуч від кнопки «Пуск») і виконавши пошук рядка на кшталт «Изменение переменных среды текущего пользователя».

2.3 Настроювання MPJ Express у конфігурації для різних кластерів

Якщо у користувачів є можливість і вони бажають виконувати свої паралельні Java-застосування на розподілених платформах пам’яті, включаючи кластери або мережі робочих станцій, то програмне забезпечення MPJ Express може бути налаштовано у конфігурації кластера.

Коли конфігурація кластера або мережі комп’ютерів відповідає рис. 2.9 а)

(тобто типовий кластер з розподіленою пам'яттю), відповідно налаштоване ПЗ MPJ Express запускає по одному своєму процесу на кожному обчислювальному вузлі, які взаємодіють один з одним за допомогою передачі повідомлень.

Але сучасні високопродуктивні кластери в основному оснащені багатоядерними процесорами (рис. 2.9 б). В цьому разі MPJ Express може ефективно і прозоро використовувати, як багатоядерну конфігурацію для зв'язку всередині вузла, так і кластерну конфігурацію для зв'язку між вузлами, відповідно.

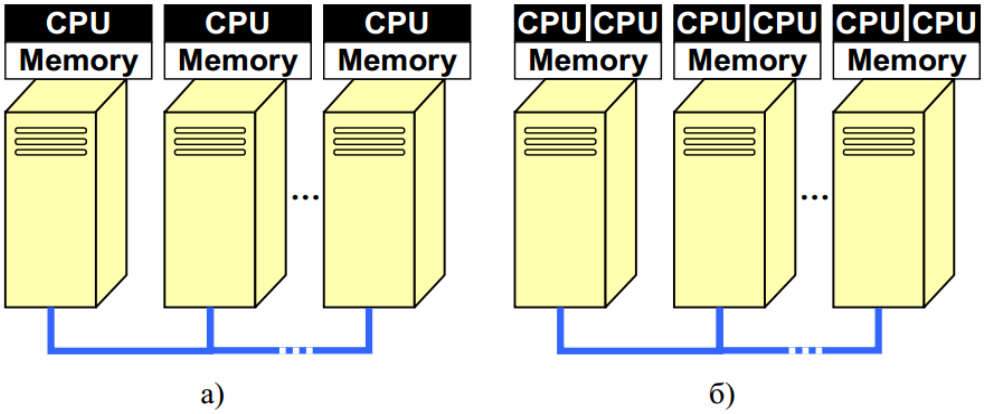


Рисунок 2.9 – Різні структури кластерів з якими може працювати MPJ Експрес а) – структура з розподіленою пам'яттю, включаючи кластери і мережі комп'ютерів; б) – гібридна конфігурація кластера з багатоядерних машин

Взагалі існує чотири комунікаційних драйвери для кластерних конфігурацій з яких розробники застосувань, за допомогою ключів командного рядка, можуть вибирати, той чи інший, відповідно до своїх потреб.

Це таки драйвери:

1. Java New I/O (NIO) драйвер, на ім'я niodev;
2. драйвер мережі Myrinet, відомий як mxdev;
3. драйвер гібридного обладнання, відомий як hybdev;
4. споріднений драйвер, на ім'я native

Драйвер обладнання Java NIO (`niodev`) може бути використаний для виконання MPJ Express програм на кластерах або у мережі комп'ютерів. Цей драйвер для передачі повідомлень використовує мережу міжз'єднань на основі

Ethernet. Оскільки багато сучасних високопродуктивних кластерів обладнані мережею Myrinet з низьким рівнем затримки повідомлень, MPJ Express для передачі повідомлень в таких мережах може використовувати драйвер обладнання `mxdev`. Цей драйвер реалізується з використанням бібліотеки Myrinet eXpress (MX) компанії Myricom.

Для виконання користувачем MPJ Express програм у конфігурації кластера, яка проказана на рис. 2.9 б) призначений спеціальний гібридний драйвер `hybdev`. Цей драйвер реалізує багатоядерну і кластерну конфігурацію для зв'язку всередині вузла і конфігурацію кластера (тільки NIO пристрій) для міжвузлового зв'язку.

Четвертий, тобто споріднений драйвер – `native`, застосовується у випадках коли паралельні Java-застосування, які написані з допомогою класів пакету MPJ Express, вимагають використання споріднених реалізацій MPI для спілкування. За допомогою цього драйвера основна частина логіки обміну повідомленнями лягає на базові бібліотеки стандарту MPI, які написані мовою C або C++. Такий підхід має рацію тому, що програми з методами класів MPJ Express (як говорилось вище це стандарт MPI-1.1) можуть використовувати новітні функції споріднених бібліотек стандартів MPI-2.0 і, навіть, MPI-3.1, таких як підтримка нових міжз'єднань і ефективні алгоритми колективних комунікацій.

Настроювання MPJ Express у всіх можливих кластерних конфігураціях попередньо вимагає тих самих дій, що і для багатоядерній конфігурації, тобто встановлення відповідних змінних оточення операційного середовища. Але у подальшому є деякі відмінності в налагоджуванні тієї чи іншої конфігурації з чотирьох можливих.

2.3.1 Кластерна конфігурація з драйвером niodev

В такий конфігурації, крім ініціалізації змінних оточення (див. підрозділ 2.2), для обох ОС Linux і Windows у каталозі користувача в будь-якому текстовому редакторі необхідно ще створити файл на ім'я «`machines`». В цьому файлі у довільному порядку в один стовпчик поміщається перелік машин на яких пе-

редбачається запуск паралельних процесів. Тобто структура файла повинна мати такий вигляд:

```
<machine_1>
<machine_2>
…
<machine_N>.
```

Тут нетермінальні символи `<machine_1>`, `<machine_2>` і т.д. повинні бути заміщені відповідними IP адресами комп'ютерів, їхніми повними іменами або аліасами. Цей файл у подальшому буде використаний системними скриптами, наприклад, такими як `mpjboot`, `mpjhalt`, `mpjrun.bat` і `mpjrun.sh` з метою визначення машин для контактів.

Після створення файла за допомогою скриптів

```
sudo mpjboot machines
```

для Linux та

```
mpjdaemon.bat -boot
```

для Windows запустити демонів для керування комунікаційною мережею. Цей скрипт по протоколу SSH передається на кожну машину, що включена до файлу `machines` і запускає на ній демона. Якщо у системі є дозвіл на ведення журналу, то кожен демон створює файл звіту з ім'ям `<machine_name>.log` в каталозі `$MPJ_HOME/logs` (ОС Linux) або у ОС Windows – `daemon-<machine_name>.log` у тому ж каталозі `%MPJ_HOME%\logs`.

Робота демонів на комп'ютерах у мережі зупиняється виконанням скриптів:

```
mpjhalt machines
mpjdaemon.bat -halt
```

для операційних систем Linux і Windows відповідно.

В загалі кажучи, немає потреби зупиняти демонів кожен раз після виконання паралельної програми. Якщо програма завершилася коректно, то нове розрахункове завдання може бути виконане без перезапуску демонів. Їхнє зупинення потребується тільки після завершення виконання всіх завдань.

2.3.2 Конфігурування кластера з драйвером mxdev

Як вже було сказано вище, ПЗ MPJ Express здатне працювати і на кластерах на базі мережі Myrinet. В цьому випадку пакет має спеціальний комунікаційний драйвер, який запускається поверх утиліт бібліотеки Myrinet eXpress (MX). При налагоджуванні роботи паралельних програм у такому кластері крім створення файла «`machines`» і запуску демонів, слід зробити декілька додаткових дій.

По-перше необхідно командою

```
sudo export MX_HOME=/opt/mx
```

додати нову змінну оточення. В команді передбачається, що драйвери бібліотеки Myrinet eXpress (MX) розташовані у каталозі `/opt/mx`. Ініціалізацію цієї змінної, як і інших, можна зробити додавши команду експорту до файла .bashrc. На наступному кроці треба відредагувати файл build.xml у каталозі $MPJ_HOME змінивши рядок

```
<target name="all" depends="compile,jars,java-docs,clean">
```

на рядок

```
<target name="all" depends="compile,mxlib,jars,java-docs,clean">.
```

Як видно, в рядку просто до атрибуту «`depends`» додано значення «`mxlib`». Після збереження файла, знаходячись у поточному каталозі

$MPJ_HOME треба запустити команду «ant» для створення додаткових бібліотечних файлів. Не дивлячись на те, що компілятор gcc виведе на консоль декілька дивних попереджень, у каталозі $MPJ_HOME/lib будуть створені додаткові споріднені бібліотеки *.so та додаткові архівні файли *.jar.

Залишається лише створити файл. В основному це виконується шляхом запуску команди з пакету Myrinet eXpress: «mx_info», в релультаті роботи цього скрипту створюється файл приблизно наступного змісту:

```
MX Version: 1.1.7rc3cvs1_1_fixes
MX Build: @indus1:/opt/mx2g-1.1.7rc3 Thu May 31 11:03:00 PKT 2007
2 Myrinet boards installed.
The MX driver is configured to support up to 4 instances and 1024 nodes.
[ .. ]
ROUTE COUNT INDEX MAC ADDRESS HOST NAME P0
----- ----------- --------- ---
0) 00:60:dd:47:ad:7c indus1:0 1,1
1) 00:60:dd:47:ad:68 indus4:0 1,1
[ .. ]
```

Файл створюється у залежності від тих машин, які мають Myrinet.

2.3.3 Настроювання MPJ Express в конфігурації з драйвером hybdev

Виконання паралельної програми на кластері з гібридною конфігурацією використовує, як багатоядерну, так і кластерну конфігурації. Тобто настроювання кластера с драйвером `hybdev` виконується так само, як і для кластера с драйвером `niodev` і повністю залежить від того чи вірно виконані дії, що описані вище у підрозділі 2.2 і пункті 2.3.1. Тобто звичайним образом встановлюються значення змінних оточення `MPJ_HOME` і `PATH`, після чого створюється файл `machines` з зазначенням тих машин, на яких буде виконуватись паралельна програма. На цьому, власне, настроювання і завершується.

2.3.4 Конфігурування кластера з драйвером native, з використанням бібліотеки MPI

Настроювання для створення та запуску паралельних програм на основі пакету MPJ Express на кластері у конфігурації зі спорідненим `native` драйвером, окрім встановлення змінних оточення за підрозділом 2.2 потребує ще й додаткових дій.

Спочатку слід переконатися, що в системі кластера встановлена і безпомилково працює яка-небудь споріднена бібліотека MPI. Якщо встановлення такої бібліотеки не було зроблено раніш, то потрібно провести завантаження та встановлення бібліотеки керуючись вказівками відповідного розробника. Наприклад, завантаження спорідненої бібліотеки MPICH можна зробити з Інтернет сторінки за адресою http://www.mpich.org/downloads. При цьому бажано слідкувати що б споріднена бібліотека, яка завантажується, була сумісною з встановленим пакетом класів MPJ Express. На момент написання посібника, пакет MPJ Express версії 0.44 тестувався розробниками на відповідність з наступними спорідненими MPI бібліотеками:

- MPICH 3.0.4
- MVAPICH 2.2
- Open MPI 1.7.4
- MS-MPI (тільки для Windows)

Хоча розробники і сподіваються, що пакет зі спорідненим драйвером буде працювати з будь-якою спорідненою бібліотекою, але якщо назва або версія спорідненої бібліотеки не відповідає названим вище, то обов’язково слід зробити ретельне тестування сумісності.

Наступним обов’язковим кроком з настроювання пакету в native-конфігурації є компіляція спорідненої бібліотеки функцій з використанням спорідненого Java-інтерфейсу – JNI.

У операційній системі Linux це робиться таким чином.

Перевіряється чи є у системі утиліта `cmake` версії 2.6 або вище, в разі не-

обхідності – встановлюється. Створюється і робиться поточним каталог `build`. Виконується це, наприклад, командою системи:

```
cd $MPJ_HOME/src/mpjdev/natmpjdev/lib; mkdir build.
```

Наступною командою с використанням CMake

```
cd build; cmake
```

генерується `makefile`. І на завершення, утилітою `make` без параметрів спочатку компілюється, а потім с параметром `install` (командою `make install`) в каталозі $MPJ_HOME/lib встановлюється загальнодоступна JNI-бібліотека на ім'я `libnativempjdev.so`.

Як вказано вище, під операційною системою Windows пакет класів тестувався лише зі спорідненою MPI бібліотекою MS-MPI розробниками якої є програмісти фірми Microsoft. Тому уразі іншої бібліотеки також необхідно провести ретельне тестування пакету MPJ Express з цією бібліотекою.

Для компіляції спорідненої бібліотеки функцій під ОС Windows потрібна наявність у системі інтегрованої системи розробки фірми Microsoft – MS Visual Studio.

Виконується це так.

Запускається Visual Studio і створюється Win32 проект, якому привласнюється ім'я «`nativempjdev`». Далі у вікні параметрів проекту (Application Settings) вказується тип застосування DLL – динамічно зв'язувана бібліотека і в розділі додаткових параметрів встановлюється прапорець «Пустий проект» (`Empty project`). На цьому створення проекту завершується. На панелі «Solution Explorer» після клацання правою кнопкою миші на створеному проекті `nativempjdev`, з меню обирається пункт «Властивості» («Properties»). У наступному діалоговому вікні «nativempjdev Property Pages» у розділі «Configuration Properties → C/C++ → General» вибрати пункт «Additional Include Directories» і у відповідному вікні вибрати каталог з додатковими файлами з комплекту роз-

робки програм мовою Java, які будуть підключатись до проекту. Наприклад, це може бути каталог D:\Program Files\Java\jdk1.7.0_09\include\win32. Далі у розділі «Configuration Properties → Linker → General» вікна «nativempjdev Property Pages» вибрати пункт «Additional Library Directories» і вказати шлях до бібліотечних файлів з пакету MS-MPI, які теж є необхідними для підключення до проекту. У якості такого шляху може слугувати, наприклад, D:\Program Files\MPI\Lib\i386. А з розділу «Configuration Properties → Linker → Input», користуючись пунктом «Additional Dependencies» задати додаткові залежності, вибравши з пропонованого переліку бібліотечний файл `msmpi.lib`. Наступним кроком є підключення до проекту заголовних файлів (*.h) від пакету класів MPJ Express. Для цього виконуються наступні дії: на панелі Solution Explorer правою кнопкою миші робиться клацання на пункті «Header Files» створеного проекту `nativempjdev` і через низку меню Add → Existing Item… у відкритому файлері відмічаються всі *.h файли з каталогу %MPJ_HOME%\src\mpjdev\natmpjdev\lib. Потім знов у панелі Solution Explorer правою кнопкою миші робиться клацання вже на пункті «Source Files» і так само у тому ж каталозі вибираються всі *.c файли. Тим самим повністю завершується наповнення порожнього проекту і його можна вже компілювати. Для компіляції достатньо правою кнопкою миші клацнути на `nativempjdev` і вибрати пункт локального меню «Build». Це призведе до створення бажаної динамічної бібліотеки `nativempjdev.dll`. Заключним кроком є копіювання одержаного файла з каталогу де він був створений (це може бути project\folder\x64\Release, можливо project\folder\x32\Release, \x64\Debug і т.д) у каталог %MPJ_HOME%\lib.

2.4 Компіляція і виконання паралельних MPJ Express програм

Як для багатоядерного режиму, так і для запуску паралельної програми в будь-якій кластерній конфігурації компіляція програми виконується однаково. Для компіляції лише потрібно в командному рядку консолі тієї чи іншої операційної системи виконати утиліту SDK Java `javac` з відповідними ключами і ім’ям файлу з вихідним кодом. Таким чином синтаксис команди компіляції для

Linux системи буде таким:

```
javac -cp .:$MPJ_HOME/lib/mpj.jar <Ім’я_програми>.java,
```

а для ОС Windows буде мати вигляд:

```
javac -cp .;%MPJ_HOME%/lib/mpj.jar <Ім’я_програми>.java.
```

Тобто компілятор Java викликається з ключем `-cp` (`-classpath`) у якому визначається шлях до бібліотеки класів пакету MPJ Express. При цьому створюється файл на ім’я `<Ім’я_програми>.class`, придатний для виконання на віртуальній машині Java.

А от командній рядок, потрібний для запуску одержаного файла `*.class`, дещо відрізняється один від одного у залежності від конфігурації обчислюваної системи, на якій буде здійснюватись запуск програми.

Запуск програми на виконання здійснюється за допомогою скриптів `$MPJ_HOME/bin/mpjrun.sh` для Linux і `%MPJ_HOME%\bin\mpjrun.bat` для Windows.

Правила підстановки, які відповідають формулам Бекуса-Наура, і за якими здійснюється побудова нетермінального символу `<Командний_рядок>` для запуску паралельної програми на виконання, наведені нижче.

```
<Командний_рядок> ::= mpjrun.<Суфікс> -np <N_проц> [<Конфіг>]
                      [<Myrinet>] <Ім’я_програми>
<Суфікс> ::= sh | bat
<N_проц> ::= <ціле_число>
<Ціле_число> := <Цифра> | <Цифра><Ціле_число>
<Цифра> := 0 | 1 | 2 |...| 9
<Конфіг> ::= -dev <Драйвер>
<Драйвер> ::= niodev | mxdev | hybdev| native
<Myrinet> ::= -Djava.library.path=$MPJ_HOME/lib
              [-mxboardnum <Дескріптор>]
<Дескріптор> ::= <Ціле_число>
<Ім’я_програми>::= <Ідентифікатор>
<Ідентифікатор> := <Літера> {<Літера>|<Цифра>}
<Літера> := _ | A | B | C |...| Z | a | b |...| z
```

Таким чином, використовуючи ці правила можна створити командний рядок для запуску відкомпільованої програми у будь-якій конфігурації кластеру під керуванням будь-якої операційної системи. Так, наприклад, командний рядок для запуску програми HelloWorld.class у багатоядерній конфігурації для операційної системи Linux з використанням п'яти процесів у найпростішому випадку виглядає так:

```
mpjrun.sh -np 5 HelloWorld,
```

а для ОС Windows, наступним чином:

```
mpjrun.bat -np 5 HelloWorld.
```

Обидва скрипти запускаються з ключем `-np` після якого слідує нетермінальний символ `N`, який замінюється числом процесів на яких буде запущена програма. Нетермінальний символ `Ім'я_програми` – це ім'я класу, що буде запущений на виконання, але без зазначення суфіксу `.class`.

Запуск тієї ж програми у Linux системі на кластері з шести процесорів з використанням драйверу Java NIO виконується командою:

```
mpjrun.sh -np 6 -dev niodev HelloWorld.
```

Якщо кластер з тієї ж кількістю процесорів під керуванням ОС Windows призначений для запуску програм зі спорідненими бібліотеками, то слід подати таку команду:

```
mpjrun.bat -np 6 -dev native HelloWorld.
```

У конфігурації мережі з устаткуванням Myrinet команда буде наступною:

```
mpjrun.sh -np 6 -dev mxdev -Djava.library.path=$MPJ_HOME/lib HelloWorld.
```

Ця команда передбачає використання мережевого адаптера Myrinet з id = 0, умовчання може бути змінено використанням ключа «`-mxboardnum`» з визначенням id адаптера.

Слід пам'ятати, що у всіх випадках кластерної конфігурації перед запуском програми на виконання слід обов'язково виконати скрипт `mpjboot` з параметром `machines` або скрипт `mpjdaemon.bat -boot` для ініціювання демонів мережі.

2.5 Контрольні запитання та завдання для самостійної роботи

– До якого типа програмного забезпечення (комерційного, вільного або іншого) відноситься пакет MPJ Express? Назвіть і охарактеризуйте ліцензію за якою він розповсюджується.

– Які недоліки описуваного пакета реалізації стандарту MPI Ви можете назвати? Наскільки критичні ці недоліки?

– Опишіть процес отримання і встановлення пакету програмного забезпечення MPJ Express.

– У яких режимах можуть працювати програми, що створені з використанням методів пакета класів MPJ Express?

– Що таке «Споріднена бібліотека MPI»? З якими спорідненими бібліотеками гарантується безпомилкова робота програм, що використовують класи пакета?

– Опишіть можливі конфігурації обладнання, які забезпечуються у програмах на основі MPJ Express.

– Розкажіть про процес настроювання пакету для роботи у багатоядерному режимі у системах відкритого типа. Які змінні середовища необхідно встановити в системі?

– А як це робиться у операційній системах сімейства Windows?

– Які комунікаційні драйвери для кластерних конфігурацій поставляються з пакетом класів? Дайте їхню стислу характеристику.

– Чим відрізняється настроювання пакету для роботи в кластерній конфігурації з драйвером `niodev` від настроювання у багатоядерному режимі?

– Напишіть файл `machines` для семи довільних комп'ютерів доступної Вам локальної мережі. Необхідні дані візьміть у системного адміністратора мережі.

– Розкажіть як виконується запуск і як зупиняється робота драйверів у кластерній системі з драйвером `niodev` у різних ОС?

– Стисло опишіть процес конфігурування пакету із спеціальним комунікаційним драйвером на кластерах на базі мережі Myrinet.

– Як налаштовується MPJ Express в конфігурації з драйвером `hybdev`?

– Яких дій потребує настроювання пакету для створення та запуску програм на кластері у конфігурації зі спорідненим native драйвером?

– Як у різних ОС виконується компіляція спорідненої бібліотеки функцій з використанням спорідненого Java-інтерфейсу?

– Розкажіть про компіляцію і запуск паралельних Java-програм для роботи у різних ОС та під керуванням різних комунікаційних драйверів.

– Опишіть синтаксис командного рядка системи для запуску Java-програми з використанням пакету класів MPJ Express.

– Напишіть всі можливі командні рядки для запуску умовної програми TestMPJ.java для обох ОС і всіх конфігурацій комунікаційної мережі з 18-ю процесами та деякими параметрами програми.

3 ІНТЕГРОВАНЕ СЕРЕДОВИЩЕ РОЗРОБКИ ECLIPSE ЯК ІНСТРУМЕНТ ДЛЯ НАПИСАННЯ Й НАЛАГОДЖЕННЯ MPJ EXPRESS ПРОГРАМ

Даний розділ присвячений розгляду питань пов'язаних з тим, як програми, які використовують бібліотеку класів MPJ Express, можуть створюватися, запускатися й налагоджуватися в IDE Eclipse.

3.1 Створення програми у середовищі Eclips

Як звичайно процес написання програми в середовищі Eclipse починається зі створення проекту. Проект може бути створений будь-яким з способів, які досить докладно описані в [14]. Наприклад, після запуску Ecipse у головному меню середовища програмування треба вибрати «File»→«New»→«Java Project» (рис. 3.1). У діалоговому вікні «Create a Java Project», що відкривається, у відповідному рядку вводу «Project name» необхідно вказати ім'я нового Java-проекту (ім'я – довільне, наприклад, щось на кшталт "MulticoreDemo"). Після уведення ім'я, за замовчуванням стає активною кнопка закінчення діалогу «Finish», але в нашому випадку *необхідно вибрати кнопку продовження діалогу* «Next» і натиснути її (рис. 3.2).

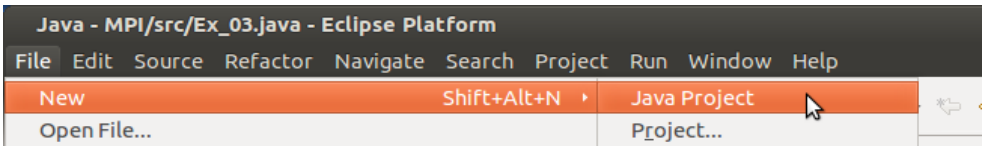


Рисунок 3.1 − Початок створення нового Java-проекту

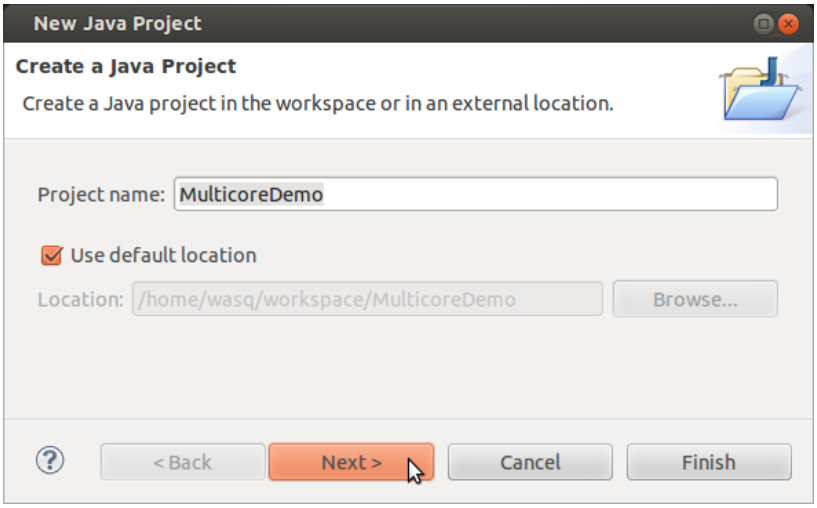


Рисунок 3.2 − Продовження створення Java-проекту "MulticoreDemo"

При цьому з'являється нове діалогове вікно з ім'ям «Java Settings». У цім вікні можна здійснити додавання зовнішніх JAR-файлів у створюваний проект.

Для цього відкривається вкладка «Libraries» і натискається кнопка «Add External JARs...» (рис. 3.3). Потім у файлері вибирається й додається у проект файл JAR-архіву – mpj.jar, який знаходиться в підкаталозі lib пакета MPJ Express (див. рис. 2.1). І ось тільки тепер натискається кнопка «Finish», і новий Java-проект на ім'я "MulticoreDemo" успішно створений.

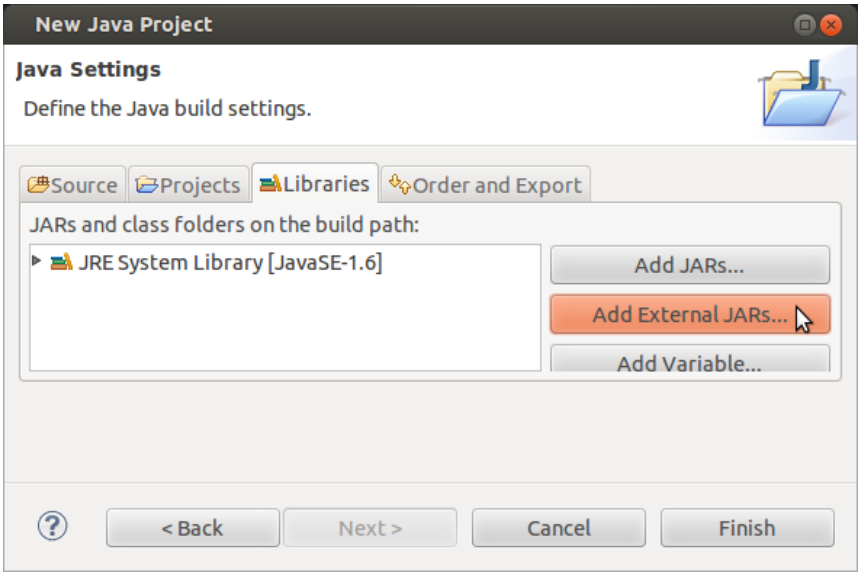


Рисунок 3.3 − Вкладка Libraries діалогового вікна Java Settings

Наступними діями, як і завжди, буде створення нового класу – «File» → «New» → «Class», наприклад, з ім'ям "HelloWorldMPJ" і написання вихідного коду класу. У якості такого коду можна написати наступне:

```
import mpi.*;
public class HelloWorldMPJ {
  public static void main(String[] args) throws Exception {
    MPI.Init(args);
    int rank = MPI.COMM_WORLD.Rank();
    int size = MPI.COMM_WORLD.Size();
    System.out.println("I am process <" + rank + "> of total <"
                       + size + "> processes.");
  MPI.Finalize();
  }
}
```

3.2 Запуск паралельних програм у Eclips

Процес запуску створеної MPJ Express паралельної програми в багатоядерному режимі усередині інтегрованого середовища складається з таких кроків.

По-перше треба зробити конфігурування створеного проекту, звернувшись до пункту «Run» головного меню і в ньому вибрати підпункт «Run Configurations…» (рис. 3.4). При цьому відкривається діалогове вікно «Run Configurations» з меткою «Create, manage, and run configurations» (рис. 3.5).

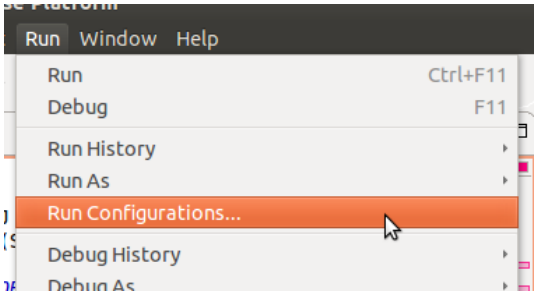


Рисунок 3.4 – Вибір вікна конфігурування проекту

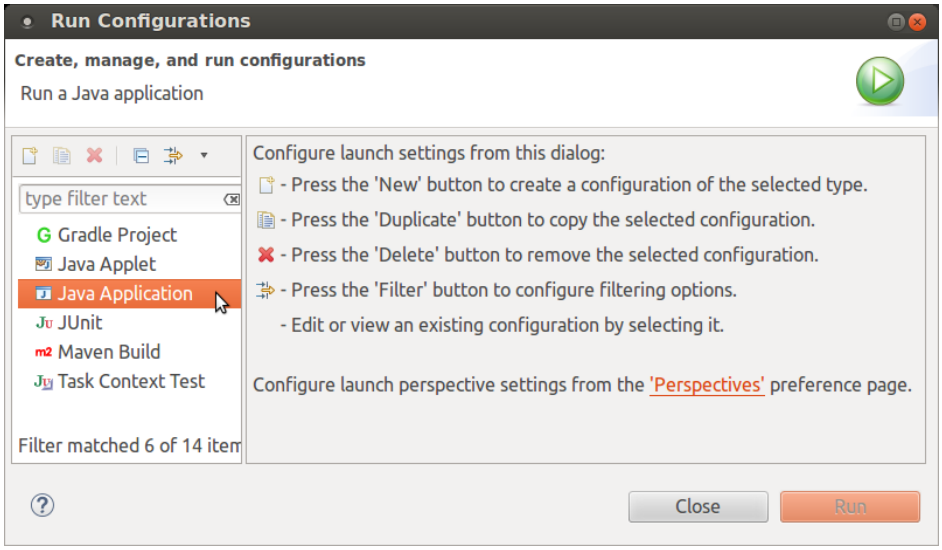


Рисунок 3.5 – Початковий вигляд вікна конфігурування проекту, вибір конфігурації запуску проекту

У відкритому вікні у лівій його частині слід вибрати тип створюваного застосування виконавши для цього подвійне клацання на елементі списку «Java

application». Результатом цих дій буде створення нової конфігурації запуску програми на ім'я "HelloWorldMPJ", що відразу відображається у правій частині вікна (рис. 3.6). Як видно з рисунку в цім вікні є низка різних вкладок, таких як «Main», «Arguments», «JRE» і т.і.

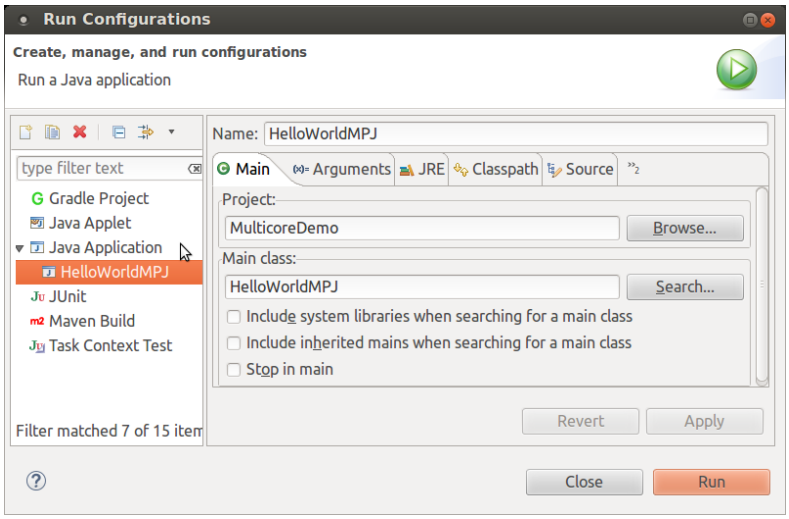


Рисунок 3.6 – Вигляд вікна конфігурування після вибору конфігурації запуску проекту

По-друге слід задати деякі параметри настроювання для правильної роботи віртуальної машини Java. Щоб зробити це, слід у тільки що відкритому вікні вибрати вкладку «Arguments». Як видно з рис. 3.7, на вкладці є два текстових поля, які призначені для вводу аргументів програми «Program arguments» і віртуальної машини Java «VM arguments».

Для виконання MPJ Express паралельної програми HelloWorldMPJ слід задати такий аргумент віртуальної машини Java:

```
-jar ${MPJ_HOME}/lib/starter.jar.
```

Але, перед тім як вводити це значення у текстове поле вводу аргументу для віртуальної машини «VM arguments», необхідно встановити змінну оточення операційного середовища – `MPJ_HOME`, якщо вона не була встановлена заздалегідь так, як це було вказано вище у пунктах 2.2.1 або 2.2.2 .

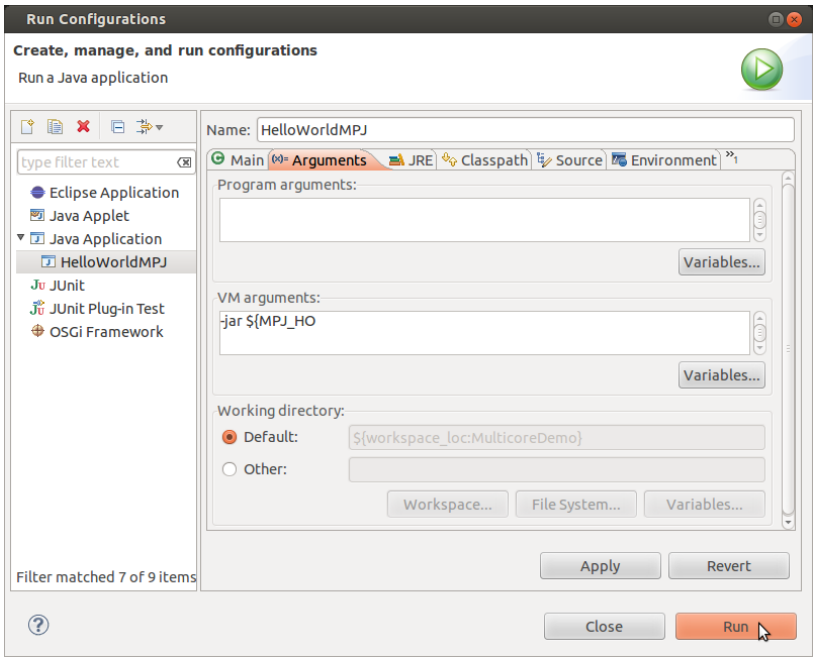
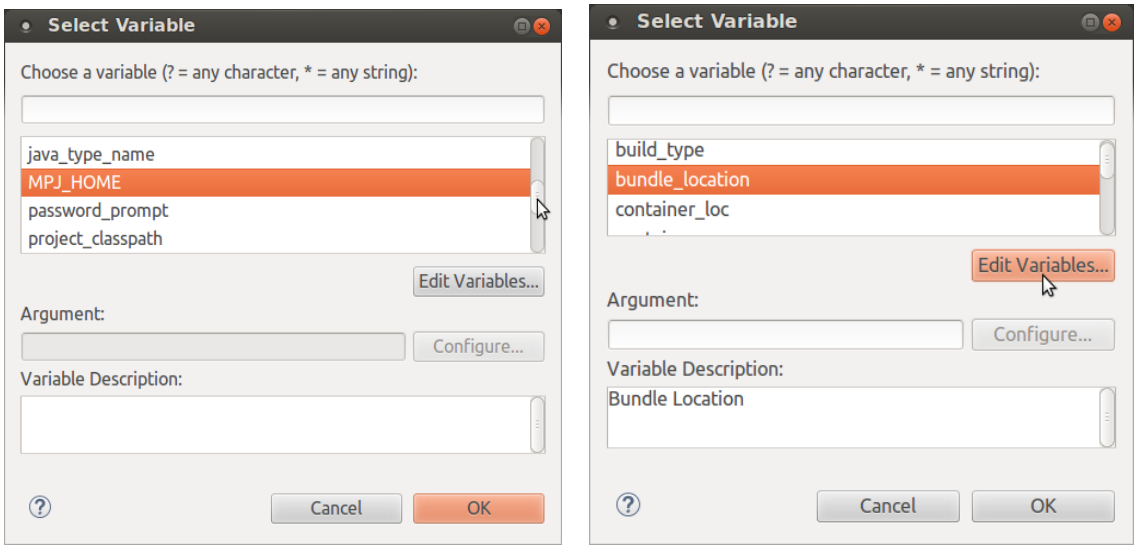


Рисунок 3.7 – Зовнішній вигляд вкладки Arguments

Для перевірки факту наявності змінної оточення `MPJ_HOME` слід клацнути по кнопці «Variables…» під текстовим полем вводу аргументу віртуальної машини. Це призведе до появи нового вікна під назвою «Select Variable» (рис. 3.8), в якому наводиться перелік всіх встановлених змінних оточення. Якщо у цім переліку присутня змінна `MPJ_HOME` (рис. 3.8 а), то просто достатньо вибрати її з переліку. У іншому разі, якщо змінна відсутня, то треба зробити клацання по кнопці «Edit Variables…» (рис. 3.8 б), що призводить до переключення у нове вікно (рис. 3.9). В цьому вікні вибирається кнопка «New», і у черговому діалоговому вікні (рис. 3.10) у полі «New» вказується ім’я змінної оточення `MPJ_HOME`, а у полі «Value» її значення – шлях до домашнього каталогу де розташовані бібліотечні файли пакета MPJ Express. Для вибору домашнього каталогу можна скористатися кнопкою «Browse…» і файлером. Коли все це зроблено, декілька разів у всіх попередньо відкритих вікнах натискається кнопка «OK».

Ну й нарешті, третій останній крок, що є необхідним для запуску програми. Коли після вибору змінної оточення знов стає видимим діалогове вікно з міткою «Create, manage, and run configurations», треба рядок ${MPJ_HOME}, що

з’явився в текстовому полі вводу аргументу віртуальної машини, доповнити так як вказано вище і відображено на рис. 3.11. Власно тепер вже можна натиснути кнопку «Run» для запуску паралельної програми.



а) б)

Рисунок 3.8 – Вигляд вікна вибору змінної оточення

а – змінна оточення MPJ_HOME раніш вже встановлена;

б – змінної MPJ_HOME у системі ще не існує

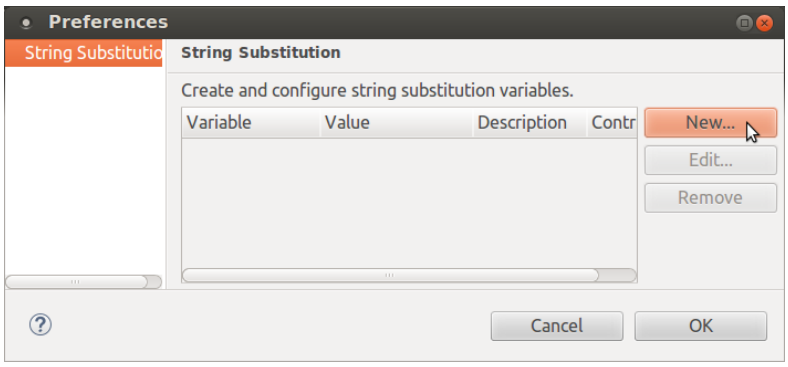


Рисунок 3.9 – Вікно створення і редагування змінних оточення

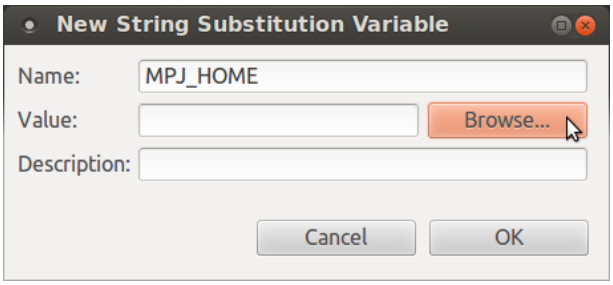


Рисунок 3.10 − Вікно встановлення параметрів нової змінної оточення

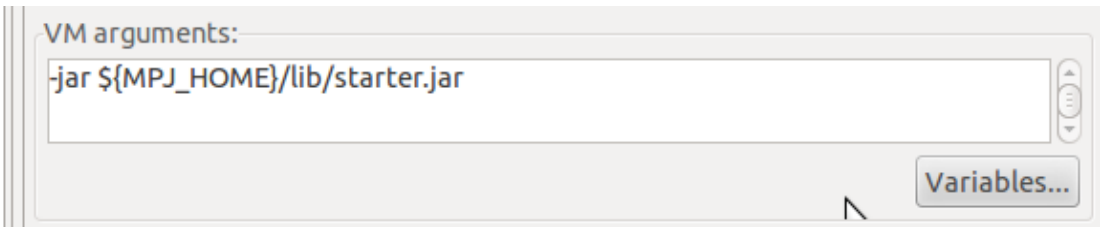


Рисунок 3.11 − Повністю введений аргумент віртуальної машини для роботи у багатоядерному режимі

Результат роботи програми відображається у віконці консолі інтегрованого середовища розробки Eclipse, яке у стандартній перспективі IDE знаходиться у низу екрана монітора. Користувач може змінити загальну кількість MPJ Express процесів, застосувавши для цього ключ «`-np`», який додається у рядок аргументу віртуальної машини Java у текстовому полі «VM arguments». Наприклад, для запуску програми з використанням п’яти ядер процесора, до аргументу віртуальної машини слід додати цей ключ зі значенням 5 тобто «`-np 5`» (без лапок). Крім того, якщо це є необхідним для вірної роботи програми, то у текстовому полі вводу «Program arguments» вводяться відповідні аргументи.

3.3 Налагодження програм за допомогою вбудованого налагоджувача

В разі необхідності, для налагодження створюваної паралельної програми у багатоядерному режимі можна скористатися вбудованим налагоджувачем IDE Eclipse. З цією метою необхідно провести відповідне настроювання параметрів інтегрованого середовища.

Для настроювання спочатку, так як описано вище, треба відкрити діалогове вікно «Create, manage, and run configurations», вибрати конфігурування запуску “HelloWorldMPJ”. Потім знов-таки відкрити вкладку «Arguments» і у текстове поле аргументів віртуальної машини «VM arguments» додати наступний рядок:

```
-agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=8000
```

Цей аргумент приписує налагоджувачу Eclipse підключитись і виконувати прослухування на порту з номером 8000 (це значення за умовчанням, але може бути змінено). По завершенні слід натиснути кнопку «Apply», а потім кнопку «Close». Наступним кроком є відкриття «Debug» перспективи IDE Eclipse [14] і установка точок переривання у вихідному коді налагоджуваної програми (у нашому випадку – HelloWorldMPJ). Тепер слід запустити програму на виконання, у консолі повинне з’явитись повідомлення приблизно такого вмісту:

MPJ Express (0.40) is started in the multicore configuration
Listening for transport dt_socket at address: 8000,

але програма не повинна запуститись на виконання.

Тепер знов треба відкрити діалогове вікно з міткою «Create, manage, and run configurations», але на цей раз за допомогою ланцюжка меню «Run» → «Debug Configurations…». У вікні подвійним клацанням вибрати пункт конфігурування запуску «Remote Java Application» для створення екземпляра конфігурації “HelloEclipseWorld (1)”, якщо така ще не існує. За замовчуванням для цього екземпляра конфігурації буде призначений порт 8000, але номер порту може бути змінений, якщо раніш у аргументі віртуальної машини був вказаний іншій номер порту. На завершення треба натиснути кнопку «Debug».

Почнеться виконання програми, але всі процеси (потоки) MPJ Express перервуть своє виконання на першій же точці переривання. Починаючи з цього моменту користувач може розпочати керування виконанням програми, вибрав-

ши вкладку «Debug» у перспективі з тим самим ім'ям. У віконці, яке має мітку «HelloWorldMPJ (1) [Remote Java Application]», відображаються параметри, які відповідають головному потоку програми. Крім того тут будуть присутні параметри інших потоків, кількість яких дорівнює кількості процесів, що запущені користувачем завдяки зазначенню ключа «-np» у аргументі віртуальної машини. Виконання цих потоків може контролюватися за допомогою різних параметрів налагоджувача, який надається середовищем Eclipse IDE.

Безпосереднє використання налагоджувача вбудованого у Eclipse IDE дозволяє налагоджувати програми лише у багатоядерному режимі на одному єдиному комп'ютері. Але починаючи з версії пакета 0.43 розробники MPJ Express поставляють спеціальний додатковий програмний продукт MPJ Express Debugger – плагін для налагодження паралельних MPJ Express програм у середовищі Eclipse.

3.4 Робота в Eclipse з зовнішнім налагоджувачем системи MPJ Express

Плагін налагоджувача системи MPJ Express ґрунтується на налагоджувачі інтегрованого середовища програмування Eclipse. Клієнт-серверна структура налагоджувача Eclipse дає користувачам можливість досить успішно виконувати налагодження, як програм, які запущені в багатоядерному режимі на окремій робочій станції, так і програми, які виконуються віддалено в багатопроцесорній мережній системі. У середовищі Eclipse у разі локального налагодження, програма запускається на конкретній робочій станції, а у разі віддаленого налагодження, програма виконується в системі, що є доступною через мережу, у якій передбачається здійснити розпаралелювання обчислювального завдання. На відміну від вбудованого налагоджувача, розглянута версія плагіна MPJ Express Debugger підтримує обидві моделі налагодження програм, як локальну, так і віддалену.

При локальному налагодженні, програма, що налагоджується, й клієнтська частина налагоджувача Eclipse запускаються на одній і тій же робочій станції, що робить сам процес такого налагодження найбільш загальним і найбільш

простим випадком налагодження. При цьому користувач може повністю використовувати весь доступний арсенал засобів налагодження програм: установку точок переривання, покрокове виконання із заходом у методи, до яких виконується звернення, зворотне покрокове трасування програми й таке інше. При локальному налагодженні й MPJ Express процес, що підлягає налагодженню, і процес налагоджувача MPJ Express Debugger повинні бути запущені на одній машині.

При віддаленому налагодженні, MPJ Express програма виконується на деякій робочій станції, а клієнт налагоджувача MPJ Express Debugger запускається на іншому процесорі в тієї ж самій локальній мережі. Віддалене налагодження є досить зручним інструментом у разі створення програми для пристроїв, які не підтримують платформу розробки. І особливо це корисно, коли виконується налагодження MPJ Express програми, яка запущена на кластері. У разі віддаленого налагодження MPJ Express програми, що виконується в багатоядерному режимі, користувач повинен указати ім'я (IP-адресу) і порт хост-комп'ютера, на якому працює застосування. У режимі ж кластерної конфігурації необхідно вказувати шлях до файлу конфігурації мережі – mpjdev.conf.

При цьому варто мати на увазі, що виконання процесу віддаленого налагодження можливо лише у разі коли встановлена в системі віртуальна Java-машина підтримує таку можливість, тобто використовується середовище виконання програм JRE версії не нижче 1.6.

3.5 Встановлення плагіну MPJ Express Debugger в інтегрованому середовищі Eclipse

Надалі буде розглянуте те, як MPJ Express програми виконуються і налагоджуються в багатоядерному і у кластерному режимі з використанням MPJ Express Debugger версії 1.0.0, яка поки що не підтримує налагодження застосувань під керуванням драйвера споріднених пристроїв. На сайті розробника за адресою http://vimeo.com/86049819 можна подивитись повний відеокурс про прийоми роботи з цим ПЗ.

Раніш ніж підключати налагоджувач MPJ Express Debugger слід, як і у підрозділі 2.1 переконатися, що встановлене у системі ПЗ сумісно з налагоджувачем. В системі повинна бути встановлена Java версії не меншої за 1.6, інтегроване середовище розробки Eclipse версії 3.6 (робоча назва «Helios») або пізніша, бібліотека класів MPJ Express версії від 0.43.

Налагоджувач може бути завантажений з офіційного сайту розробника MPJ Express або можна виконати компілювання и створення архіву MPJ_Express_Debugger.jar, використавши для цього вихідні файли налагоджувача, які знаходяться у підкаталозі debugger/src/mpj_express_debugger пакета ПЗ (див. рис. 2.1). У разі спроби компілювання, необхідно щоб у системі ще були встановлені Apache ant 1.6.2 (не менш) і Perl, які потрібні для компілювання.

Одержаний тим чи іншім способом архів MPJ_Express_Debugger.jar поміщається в каталог плагінів Eclipse, і останній запускається на виконання користувачем с правами адміністратора, це дозволяє тільки що встановленому новому розширенню Eclipse підключитися до нього. Щоб переконатися, що підключення сталося, слід відкрити вікно конфігурування IDE (ланцюжок меню «Run» → «Run Configurations…»). Наявність можливості конфігурувати запуск програми як «Express Application» (рис. 3.12) свідчить про вдале встановлення налагоджувача в середовищі.

3.6 Створення, конфігурування і запуск MPJ Express застосування в багатоядерній конфігурації

Тепер програма MPJ Express може запускатися і налагоджуватися не тільки у багатоядерному режимі, але і в конфігурації кластера, і в комбінованому режимі. Настроювання виконання програми MPJ Express у всіх конфігураціях є практично однаковим за винятком зазначення в конфігураціях різних драйверів пристроїв.

Створення нового Java-проекту (нехай це буде “MPJEclipseTest”) виконується майже так само як і в попередньому випадку включно до написання вихідного коду (див. стор. 33 – 35, рис. 3.1 – 3.3). Надалі перевірка встановлення

змінної MPJ_HOME не виконується, передбачається, що в ОС тим або іншім чином вже встановлені змінні MPJ_HOME та PATH.

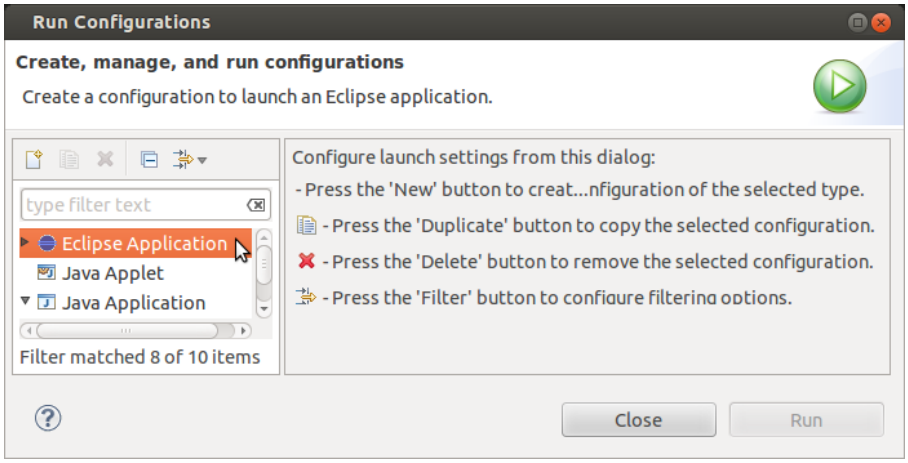


Рисунок 3.12 – MPJ Express Debugger підключений до IDE Eclipse

На цьому кроці знов-таки треба звернутися до вікна конфігурування проекту «Run Configurations». Зробити це можна або скориставшись меню «Run» (рис. 3.4), або за допомогою локального меню, клацнувши правою кнопкою миші на імені класу в «Package Explorer», як це зображено на рис. 3.13.

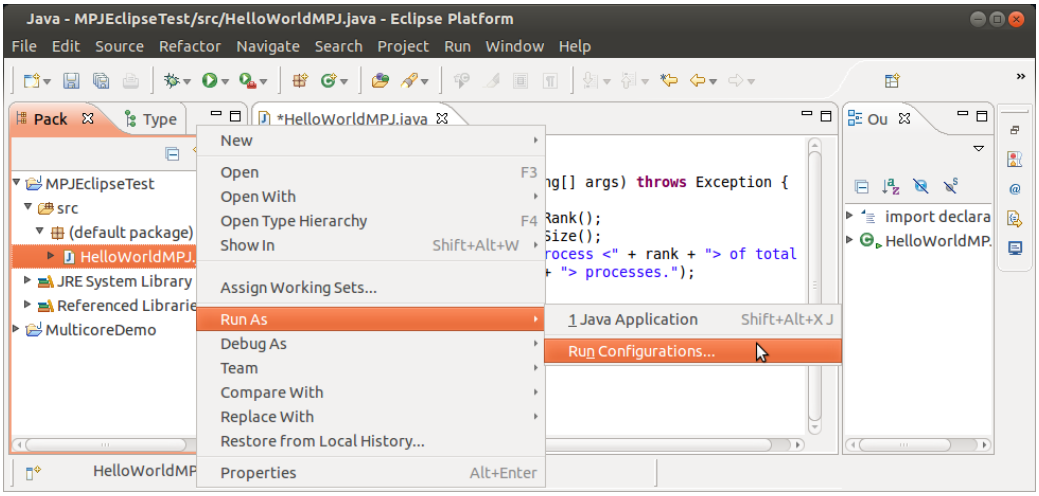


Рисунок 3.13 – Відкриття вікна Run Configurations

У вікні, що відкрилося, треба зробити подвійне клацання на новому пункті «MPJExpress Application». Результатом цього буде відкриття нової конфігура-

ції з ім'ям "HelloWorldMPJ", у якій міститься низка вкладок.

Тут вже треба вибрати вкладку «MPJ Parameters» (рис. 3.14). На цій вкладці у полі вводу «np» можна вказати бажану кількість процесів, в полі «dev» – конфігурацію обчислюваної системи (можливі значення поля: «multicore», «niodev», «hybdev») і додаткові аргументи в полі «Parameters», якщо такі потрібні.

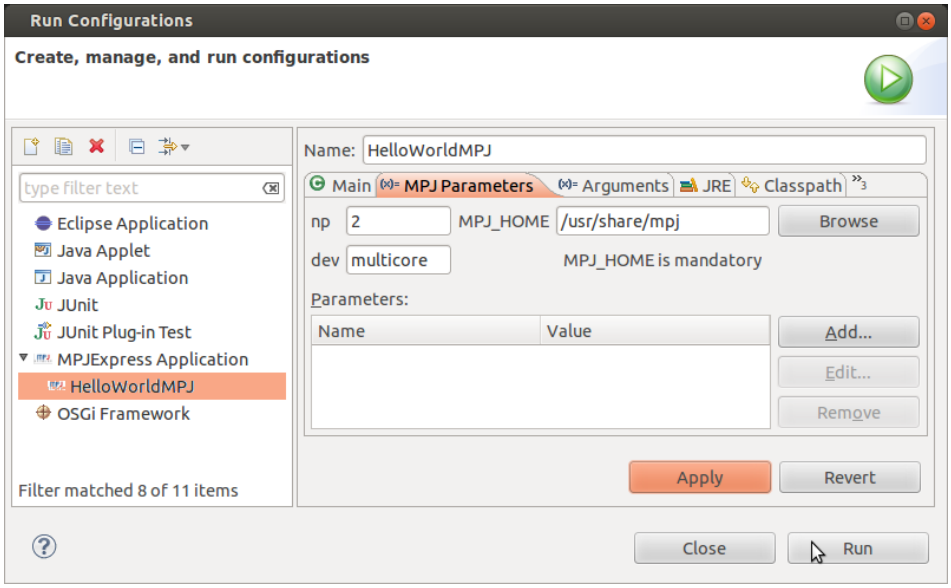


Рисунок 3.14 – Встановлення параметрів для багатоядерній конфігурації

Якщо не відбулося автоматичне заповнення поля «MPJ_HOME», то обов'язково в це поле треба ввести кореневий каталог пакету MPJ Express. Зробити це можна за допомогою файлера, після натискання на кнопку «Browse» На рисунку показана підготовлена таким чином багатоядерна конфігурація запуску паралельного застосування. На завершення потрібно послідовно натиснути кнопки «Apply» і «Run», застосування буде запущено.

3.7 Запуск MPJ Express застосування у режимі кластера

Для запуску застосування MPJ Express на кластері робочих станцій, як у конфігурації з драйвером niodev, так і комбінованому режимі с драйвером hybdev, попередньо необхідно виконати дії, які були описані вище у пункти

2.3.1. Таки дії полягають в створенні файла «machines» і збереженні його у каталозі користувача, а потім за допомогою відповідних скриптів (mpjboot для користувачів Linux та mpjdaemon.bat -boot для Windows-користувачів), знаходячись у каталозі користувача, запустити демонів тієї або іншої мережі. Осталось тільки виконати ті самі кроки, що і для запуску застосування в багатоядерному режимі, але у полі «dev» слід вказати ім'я або драйвера гомогенного кластера – niodev, або hybdev (рис 3.15) в залежності від обраної конфігурації обчислювальної мережі.

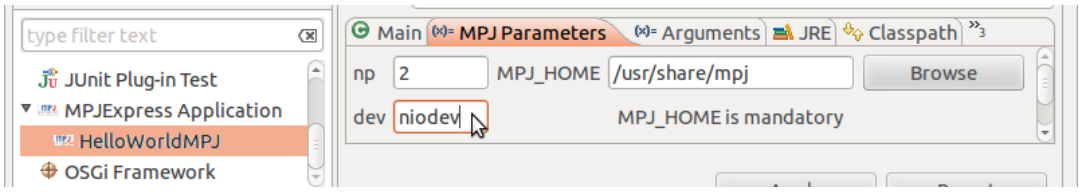


Рисунок 3.15 − Параметри запуску MPJ Express застосування у гомогенній кластерній конфігурації

3.8 Налагодження MPJ Express застосування за допомогою MPJ Express Debugger

Як вже згадувалося користувачі за допомогою MPJ Express Debugger можуть налагоджувати MPJ Express застосування в двох типах конфігурації: у багатоядерній і у кластерній. В обох випадках для запуску процесу налагодження виконуються ті ж самі дії. Налагоджувач MPJ Express Debugger в обох конфігураціях підтримує налагодження паралельних програм у двох режимах:

- локальне налагодження;
- віддалене налагодження.

3.8.1 Процес локального налагодження

Має сенс спочатку розглянути як відбувається процес налагодження у локальному режимі роботи.

В цьому режимі MPJ Express застосування створюється звичайним спосо-

бом. Тобто виконуються всі кроки до відкриття вікна конфігурування. Цього разу вікно «Create, manage, and run configuration» відкривається наступним ланцюжком пунктів меню: «Run» → «Debug Configurations». Перед цім треба встановити всі потрібні точки переривання програми. Зараз у відкритому вікні видні дві два додаткових пункти для яких можливе завдання параметрів конфігурації – «MPJExpress Application» та «Remote MPJExpress Application» (рис. 3.16).

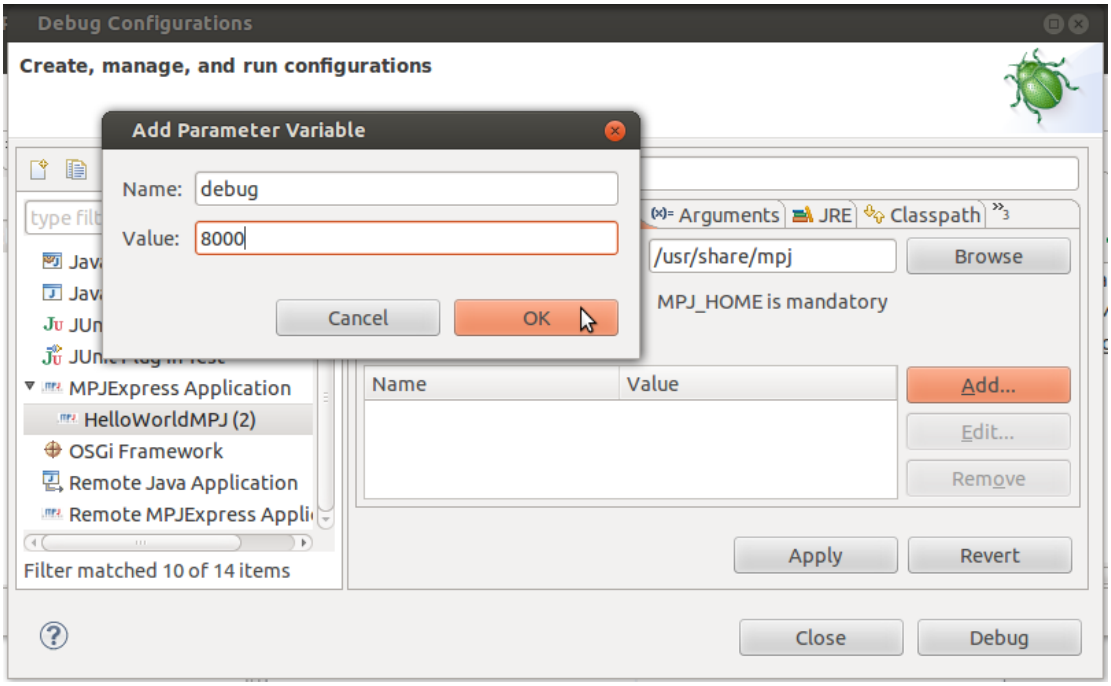


Рисунок 3.16 – Ввід параметрів MPJ при локальному налагодженні

Оскільки передбачається налагодження у локальному режимі, то слід вибрати пункт «MPJExpress Application» і зробити двійне клацання по ньому. У результаті буде створена нова конфігурація запуску з ім’ям “HelloWorldMPJ ”. Далі звичайним способом настроюється значення змінної оточення MPJ_HOME, задається потрібна кількість процесів, вказується потрібний драйвер (multicore, niodev або hybdev).

У полі «Name» вказується ім’я параметра «debug», а в полі «value» його значення «8000». Це значення відповідає порту за умовчанням на якому сервер налагоджувача буде виконувати прослухування. Якщо існує така потреба, можна вказати іншій номер порту.

На завершення натискуються кнопки «Apply» і «Debug».

В результаті відкривається перспектива середовища розробки «Debug», а всі процеси призупиняють своє виконання на точці переривання з відповідним повідомленням (рис. 3.17).

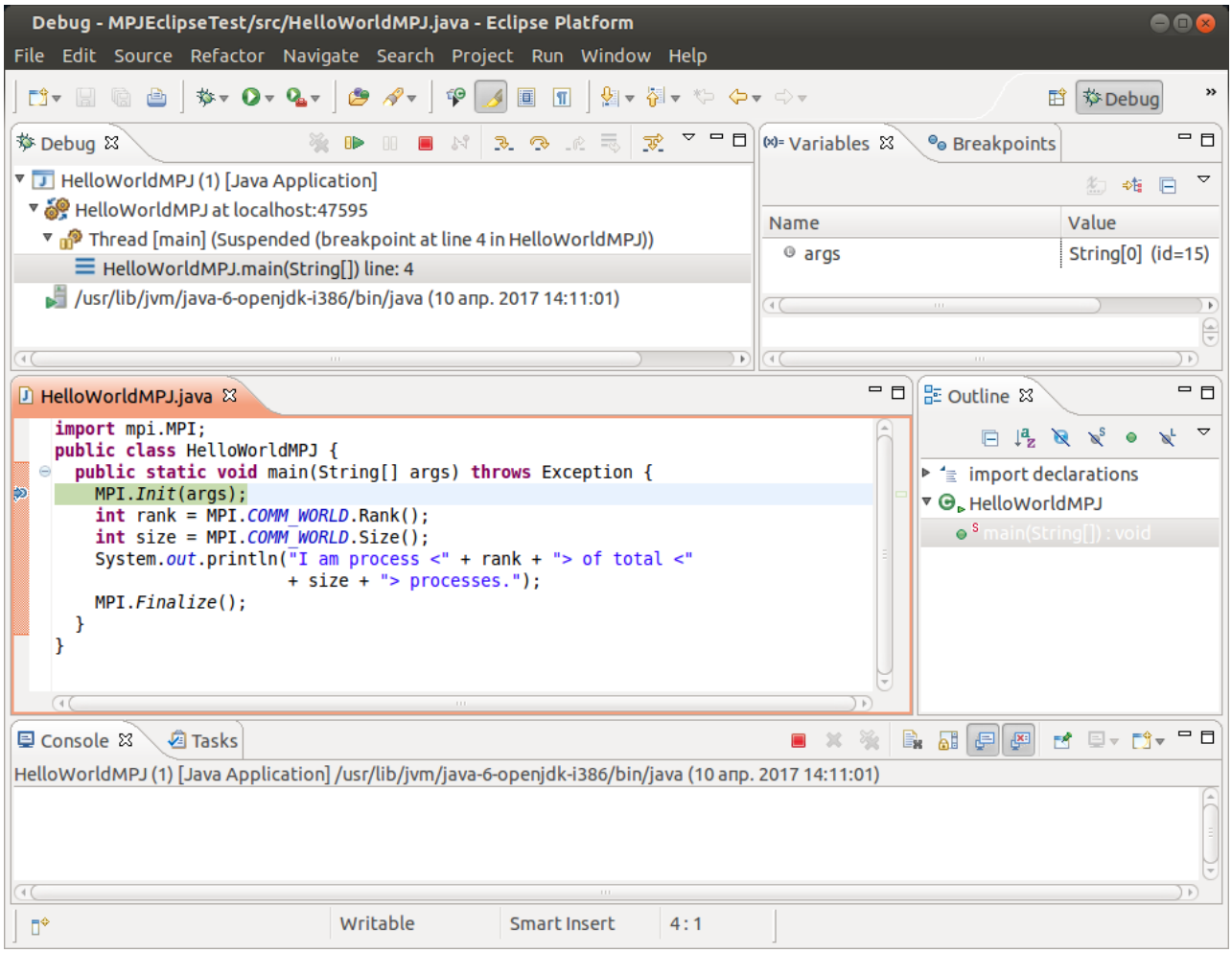


Рисунок 3.17 – Вигляд вікна перспективи Debug

Тепер є можливість вибирати будь-який потік, використовувати пряме, покрокове и зворотне трасування потоків в програмі з відстеженням стану об’єктів і, взагалі, застосовувати всю функціональність налагоджувача MPJ Express Debugger.

Цілком зрозуміло, що в разі проведення налагодження в локальному режимі, але із застосуванням драйверів niodev або hybdev, попередньо треба створити файл «`machines`» і загрузити відповідний драйвер.

3.8.2 Віддалене налагоджування

В цьому режимі MPJ Express Debugger підтримує пару окремих механізмів віддаленого налагодження:

– багатоядерне;
– розподілене.

Початкові дії для обох типів віддаленого налагодження повністю однакові і практично повторюють ті дії, що були описані вище.

Тобто стандартними прийомами у Eclipse створюється нове MPJ Express застосування. У програмі встановлюються потрібні точки переривання. Відкривається вікно «Create, manage, and run configuration» конфігурування запуску застосування (рис. 3.18) і подвійним клацанням вибирається «MPJExpress Application». Тім самим створюється нова конфігурація запуску для “HelloWorldMPJ”

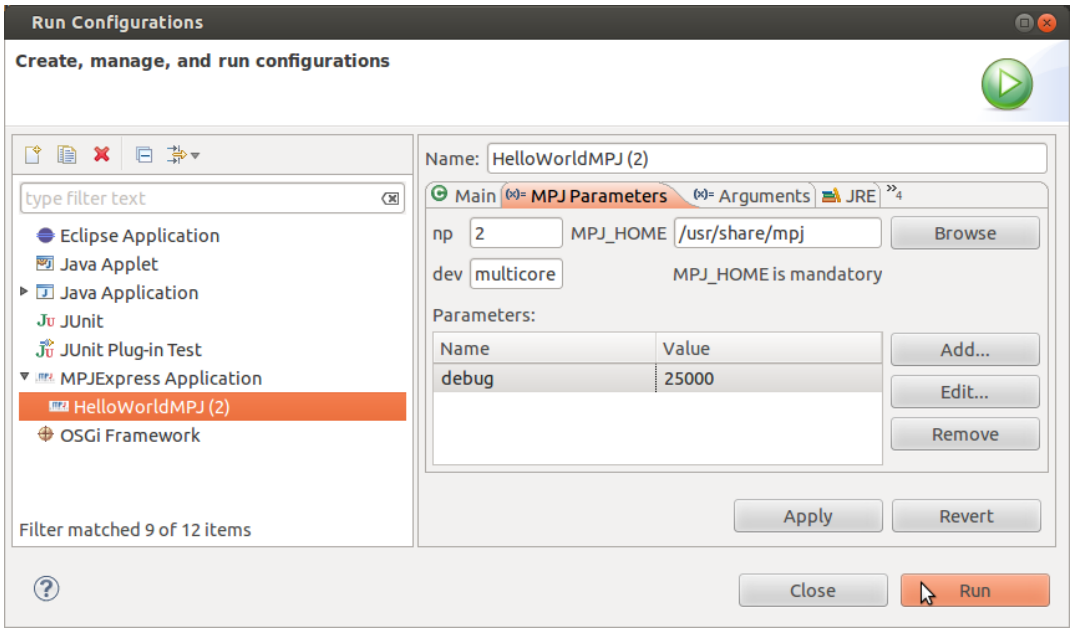


Рисунок 3.18 – Вигляд вікна завдання параметрів при віддаленому налагодженні

У полях вікна встановлюються потрібні параметри:

– шлях до кореневого каталогу MPJ Express (поле «MPJ_HOME»);
– в полі «np» кількість процесів;
– ім’я драйвера мережі (multicore, niodev, hybdev) в полі «dev»;

– додається новий параметр з ім’ям «debug» і якимсь значенням (за умовчанням 8000, але може бути змінено).

Нарешті натискується кнопки «Apply» і «Run».

Після настроювання застосування відкривається у перспективі Eclipse «Debug» саме так, як показано на рис. 3.17 з прослуховуванням порту, який був заданий в значенні параметра «debug».

Останнім кроком на цьому етапі є відкриття вікна конфігурування режиму налагодження – «Run» → «Debug Configuration» у якому подвійним клацанням на «Remote MPJExpress Application» додається нова конфігурація. У обох випадках вибирається вкладка «Connect» (рис. 3.19, 3.20).

Але у разі багатоядерного віддаленого налагоджування (рис. 3.19) для поля «Connection type» треба вибрати значення «Multicore». У полі «Host» указується IP-адреса того вузла на якому запущено MPJ Express застосування. Тут можна також вказати значення localhost. Значення порту у полі «Port» повинно повністю співпадати зі значенням порту, яке було вказано раніш при конфігуруванні запуску (рис. 3.18). Натискуються кнопки «Apply» і «Debug».

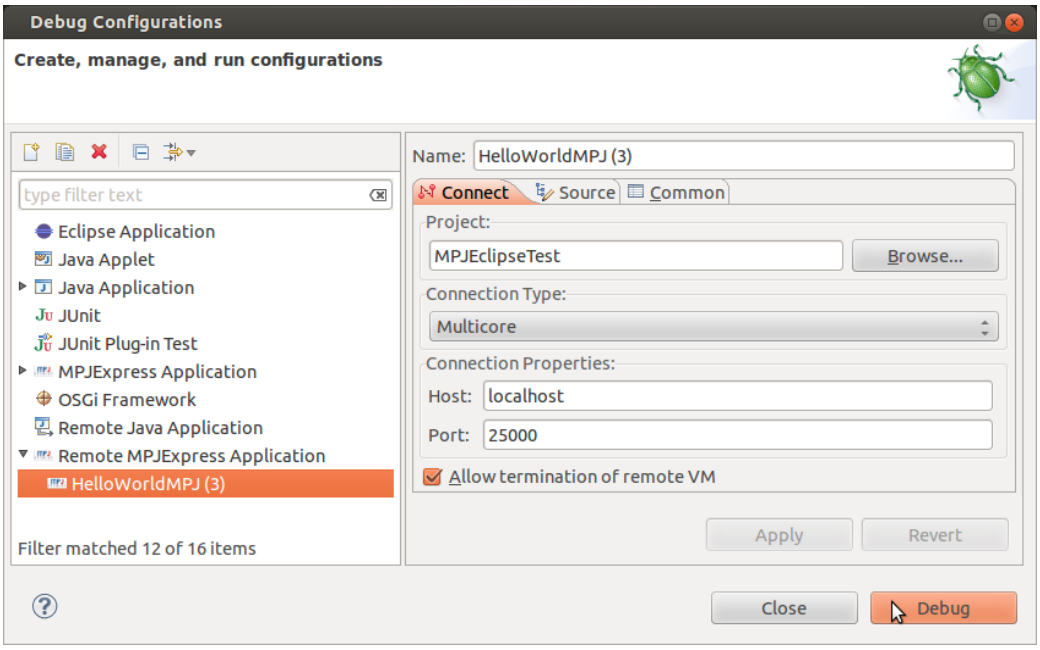


Рисунок 3.19 – Параметри зв’язку для багатоядерного режиму віддаленого налагодження

Тепер MPJ Express застосування може віддалено налагоджуватися у багатоядерному режимі.

А в разі розподіленого віддаленого налагодження (рис. 3.20) слід попередньо написати файл «`machines`» і запустити демонів керування мережею (див. п. 2.3.1).

В полі «Connection type» слід вибрати «Distributed Memory». За допомогою файлера, після натискання на кнопку «Browse» в полі «Configuration file» необхідно вказати каталог в якому MPJ Express Debugger створить конфігураційний файл mpjdev.conf. Цей файл повинен знаходитись в каталозі де міститься основний клас або JAR-файл, і до нього повинний бути вільний доступ для робочої станції, на якій запущений клієнт налагоджувача.

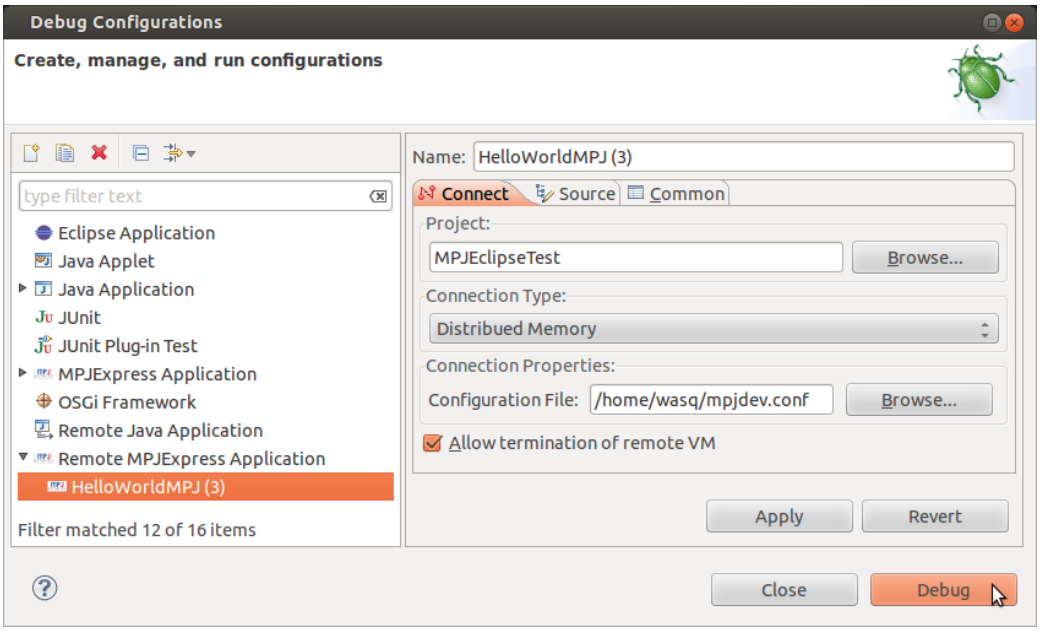


Рисунок 3.20 – Параметри зв'язку для віддаленого налагодження у розподіленому режимі

Натискуються кнопки «Apply» і «Debug».

MPJ Express застосування буде запущено в режимі віддаленого налагодження в конфігурації мережі під керуванням драйвера niodev або hybdev. Користувачу тепер доступна вся функціональність налагодження: покрокове трасування, трасування коду методів, зворотне трасування.

4 СКЛАД ПАКЕТУ, ІЄРАРХІЯ КЛАСІВ ТА ЇХНІ МЕТОДИ

Перш ніж переходити безпосередньо до розглядання питань створення паралельних програм з використанням методів пакету MPJ Express, треба добре уяснити собі ієрархію всіх класів пакету та коротко ознайомитись з приналежністю методів до тих чи інших класів. Ці знання у подальшому дозволять легше створювати і розуміти паралельні алгоритми за допомогою парадигми пересилання повідомлень.

4.1 Склад пакету і дерево ієрархії класів

На рис. 4.1 наведена UML діаграма, яка демонструє склад пакету MPJ Express. Як видно з діаграми пакет складається з п’яти підпакетів, деякі з них в свою чергу містять інші підпакети.

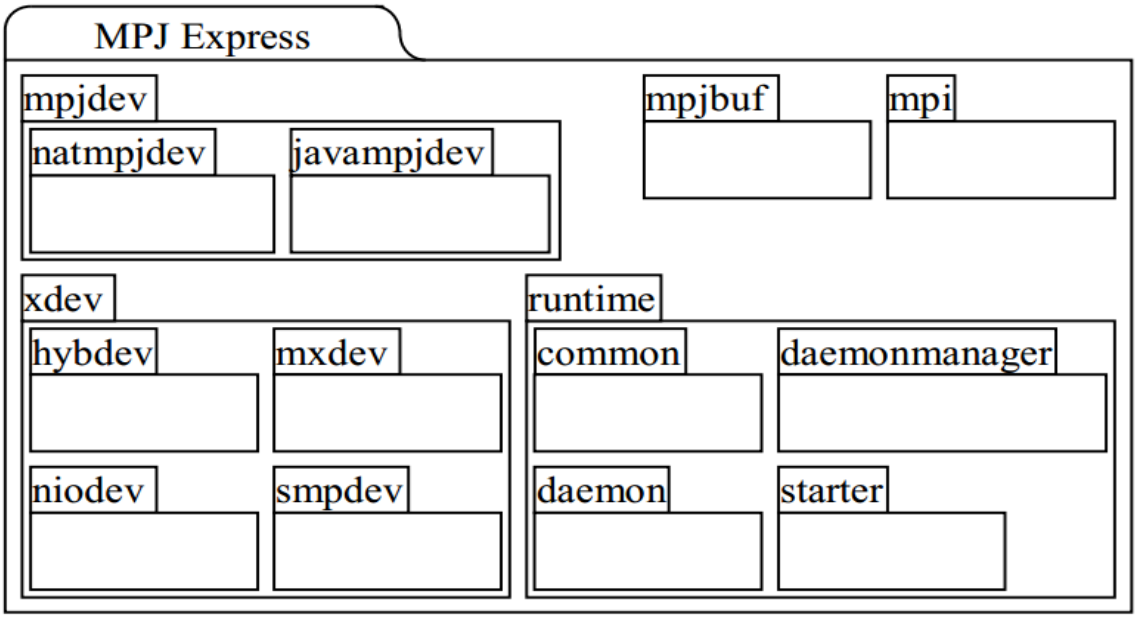


Рисунок 4.1 − UML діаграма структури пакету MPJ Express

Таки пакети як mpjdev і xdev разом зі всіма їхніми підпакетами призначені для створення драйверів різних конфігурацій кластерів, необхідність яких була розглянута вище. Пакет runtime з підпакетами підтримує створення і запуск системних демонів, а також, за допомогою класів цього пакету, безпосередньо

виконується завантаження паралельної програми на виконання. Ще один допоміжний пакет mpjbuf призначений для створення і керування буферами прийому/передачі повідомлень, які розповсюджуються у системі під час виконання програми. Переважна більшість класів пакету MPJ Express для зручності і з точки зору семантичної єдності включені до підпакету mpi (рис. 4.2).

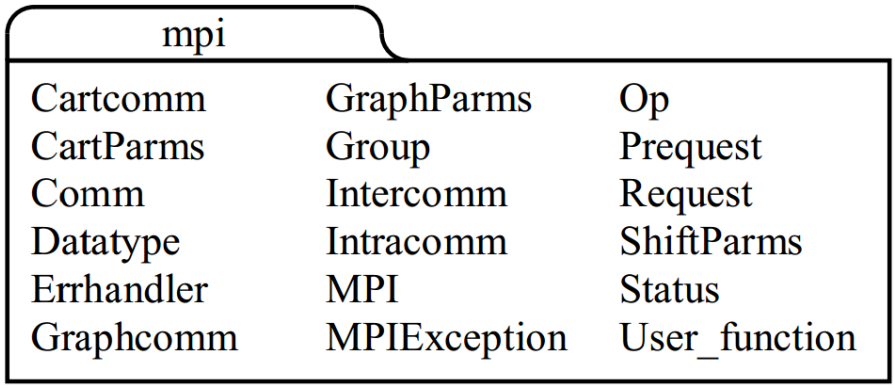


Рисунок 4.2 − UML діаграма пакету mpi

Поля і методи класів, які входять у нього, надають можливість повністю реалізувати всі ті функції, які властиві бібліотечним пакетам MPI інших розробників, і які підтримують створення паралельних програм по технології пересилання повідомлень на рівні стандарту версії 1.1. Як вже вказувалось вище, підтримка механізмів, що реалізовані у старших версіях стандарту, здійснюється за допомогою драйверів споріднених систем (native-драйвери) на кластерах де є відповідне програмне забезпечення.

Однак UML діаграма пакету надає лише відомості про складові пакету, але ні яким чином не відображає взаємовідносини між класами у пакеті. UML діаграма (рис. 4.3) наочніше пояснює відносини залежностей, які існують між класами у пакеті. З наведеного дерева видно, що переважна більшість класів пакету MPJ Express, як це й прийнято у Java, породжені від кореневого класу мови – java.lang.Object, винятки становлять лише два класи. Перший виняток це клас Status з підпакету mpjdev, але від нього походить субклас `Status` вже з підпакету mpi. І другий виняток – клас `MPJDevException` також з підпакету mpjdev, він

є суперкласом для класу винятків MPIException, який включений у підпакет mpi. Тому слід чітко розуміти і постійно пам'ятати, що, наприклад, класи Intercomm та Intracomm наслідують всі захищені й загально доступні поля і методи свого суперкласу – класу Comm. Так само класи Cartcomm і Graphcomm розширюючи клас Intracomm і являючись його субкласами повністю наслідують всі методи і поля від нього.

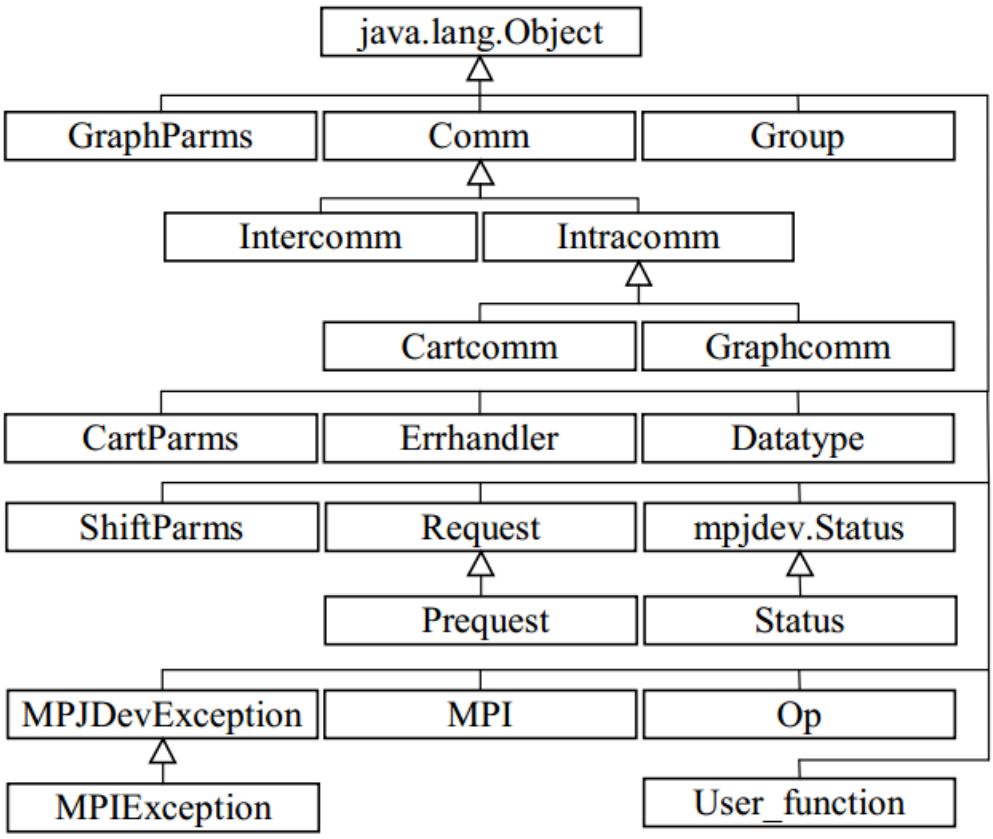


Рисунок 4.3 – UML дерево спадкування класів в пакеті mpi

Ще три дочірніх класи: Prequest, Status і MPIException також успадковують і розширюють таки власні суперкласи, як Request, mpjdev.Status і MPJDevException, відповідно.

Особливий інтерес представляє структура класу MPI (рис. 4.4). Наведена UML діаграма наочно показує, що клас через свої поля агрегує поля і методи багатьох інших класів. Ще однією особливістю класу є те, що всі його члени, як поля, так і методи, оголошені з модифікатором static, тобто, за правилами

ООП, доступ до таких членів класу, може здійснюватися лише через ім'я класу, а не через його об'єкт.

Справа в тому, що стандарт MPI передбачає використання префіксу «MPI_» перед всіма функціями і константами бібліотеки. Так, наприклад, звертання до функції відправлення повідомлення типа точка-точка у загальному комунікаторі за стандартом для мови C повинно виконуватись за допомогою ідентифікатору MPI_COMM_WORLD_Send. Ідентифікатор константи, яка асоціюється з будь-яким відправником повідомлення згідно зі стандартом має наступний вигляд MPI_ANY_SOURCE. Однак, мові Java завдяки її повній об'єктній орієнтованості таке конструювання ідентифікаторів не властиво, але стандарту дотримуватись необхідно. Названі властивості класу MPI і надають можливість, зовсім незначно порушуючи стандарт, але повністю дотримуючись синтаксису мови Java, створювати складені імена для відповідних методів і констант.

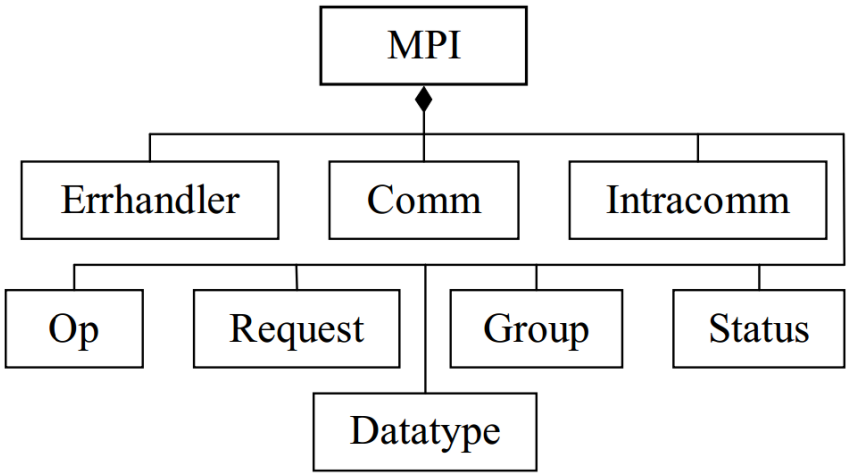


Рисунок 4.4 − Діаграма композиції для класу MPI пакету mpi

У мові Java складене ім'я того ж самого методу відправлення повідомлення має наступний вигляд: MPI.COMM_WORLD.Send. Як відомо таке ім'я відповідає тому, що повідомлення посилається об'єкту COMM_WORLD класу Intracomm. Об'єкт в свою чергу є статичним полем класу MPI. Повідомлення обробляється методом Send, який успадковується класом Intracomm від класу Comm. Аналогічно стандартне ім'я константи у мові Java перетворюється на складове ім'я ста-

тичного поля класу: `MPI.ANY_SOURCE`.

4.2 Розподіл методів класів по функціональнім групам пакету MPJ Express

Стандарт MPI мінімально передбачає, що відповідні бібліотеки повинні забезпечувати механізми для вірної безпомилкової роботи наступних функціональних розділів системи передачі повідомлень:

1) операцій і даних загального призначення;
2) комунікацій типа точка-точка;
3) засобів колективної комунікації;
4) операцій з групами;
5) віддалену комунікацію;
6) дії з віртуальними топологіями процесів.

Повна кількість функцій стандарту MPI, а відповідно і кількість методів пакету MPJ Express для забезпечення функціональності всіх цих розділів, сягає і навіть перевищує сотню. Всі ці методи розташовані у різних класах пакету mpi, успадковуються одним класом від другого. Тому є сенс яким-небудь чином упорядкувати таку велику кількість методів по ознакам приналежності методів до класів, які забезпечують функціонування того чи іншого розділу інтерфейсу передачі повідомлень. У наступних шести таблицях приведений розподіл класів з їхніми методами по групам дій, які забезпечуються ПО пакета. Основною метою цих таблиць-карт спеціальних методів пакета mpi є надання допомоги розробникам в вирішенні питання, наприклад, про те до якого класу належать методи точка-точка (або будь-які інші) в програмному забезпеченні MPJ Express.

Нижче, при описанні, власно, самої системи програмування ці методи класів будуть розглянуті більш докладно.

В табл. 4.1 наведений перелік класів і методів MPJ Express, які відповідають за виконання операцій загального характеру: ініціалізацію системи, контроль деяких параметрів, завершення роботи процесів, тощо. У цю ж групу включений такий важливий елемент системи, як «***комунікатор***».

Опис групи комунікацій типа точка-точка між процесами, відповідальні за них методи і класи пакета надані у табл. 4.2. Тут зібрані операції, які здійснюють різні алгоритми обміну повідомленнями між двома окремими процесами у кластері, а також похідні типи даних, що існують у системі.

Таблиця 4.1 − Операції групи загального призначення

Типи дій	Класи	Поля и методи
Ініціалізація	MPI	Init
Завершення	MPI	Finalize
Комунікатор	MPI	COMM_WORLD
Доступ до властивостей	MPI	Get_processor_name, Wtime, Wtick, Initialized
	Comm	Rank, Size

Таблиця 4.2 − Типи операцій з групи повідомлень «точка-точка»

Типи дій	Класи	Поля и методи
Операції з блокуванням	Comm	Send, Recv, Sendrecv, Probe, Bsend, Ssend, Rsend, Irsend, Ibsend
Операції без блокування	Comm	Isend, Irecv, Sendrecv_replace, Iprobe
	Request	Wait, Waitall, Waitany, Test, Waitsome, Cancel
Синхронні операції	Comm	Send_init, Bsend_init, Ssend_init, Rsend_init
Пустий процес	MPI	NULL_PROCESS
Похідні типи даних	MPI	BYTE, CHAR, SHORT, BOOLEAN, INT, LONG, FLOAT, DOUBLE, OBJECT, LB, UB, PACKED
Операції з буферуванням	MPI	Pack, Unpack, Buffer_attach, Buffer_detach

Наступна – третя група операцій (табл. 4.3), яки обов'язково виконуються одночасно одразу всіма процесами комунікатора забезпечується лише методами одного класу – `Intracomm`. Крім вказаних у таблиці клас також успадковує методи класу `Comm`.

Таблиця 4.3 − Група різних колективних операцій

Типи дій	Класи	Поля и методи
Різні операції	Intracomm	Barrier, Bcast, Gather, Gatherv, Scatter, Scatterv, Allgather, Allgatherv, AlltoAll, AlltoAllv, Reduce, Allreduce, Reduce_scatter, Scan

До четвертого розділу (табл. 4.4) відносяться операції, що діють з такими конструкціями стандарту MPI, як «група процесів». Створюють групи, додають і виключають процеси з групи, тощо. Функціонування забезпечується методами класів `Group` і `MPI`.

Таблиця 4.4 − Операції з групами

Типи дій	Класи	Поля и методи
Доступ до властивостей	Group	Size, Rank, Translate_ranks, Compare
Створення груп	Group	Union, Intersection, Difference, Incl, Excl, Range_incl, Range_excl, Split
Пуста група	MPI	GROUP_EMPTY
Видалення груп	Group	Free

За здійснення комунікації віддалених процесів у обчислювальних системах з розподіленою пам'яттю і створення комунікатора, який містить у собі такі процеси, що працюють на різних процесорах, здебільшого відповідають методи класу `Intercomm`, частково приймають участь методи класу `Intracomm` і методи успадковані ними від класу `Comm` (табл. 4.5).

Класи `Cartcomm` і `Graphcomm` зі всіма успадкованими методами відповідають за створення і керування так званими «***віртуальними топологіями процесів***». Такий підхід допомагає розробляти більш логічні і прости алгоритми паралельних розрахунків (табл. 4.6).

Таблиця 4.5 − Операції віддаленої комунікації

Типи дій	Класи	Поля и методи
Доступ до властивостей	Intercomm	Test_Inter, Remote_size, Remote_group,
Операції у інтеркомуні-каторі	Comm	Create_intercomm
	Intracomm	Clone
	Intercomm	Merge
Кешування	Comm	Attr_get

Таблиця 4.6 − Дії з віртуальними топологіями процесів

Типи дій	Класи	Поля и методи
Створення декартової топології	Cartcomm	Create_cart
Допоміжні функції	Cartcomm	Dims_create
Створення топології графа	Graphcomm	Create_graph
Функції топологічних запитів	Cartcomm	Topo_test, Get, Rank, Coords
	Graphcomm	Get, Neighbors
Зсув декартових коор-динат	Cartcomm	Shift
Розмітка декартових структур	Cartcomm	Sub
Топологічні функції ни-зького рівня	Cartcomm	Map
	Graphcomm	Map

4.3 Контрольні запитання та завдання для самостійної роботи

− З яких підпакетів складається пакет класів MPJ Express? Опишіть їхнє призначення.

− До якого підпакету включена переважна більшість класів пакету MPJ Express? Скільки і які класи входять у нього?

− Розкажіть про структуру класу `MPI`. Сформулюйте і поясніть основні властивості цього класу. По змозі нарисуйте UML-діаграму, що йому відповідає.

− Які функціональні розділи системи передачі повідомлень повинні хоча б у мінімальному обсязі підтримуватися відповідними бібліотеками і вірно та безпомилково працювати згідно з вимогами стандарту MPI?

− Назвіть і хоча б стисло охарактеризуйте операції загальної групи стандарту MPI.

− Які типи операцій входять у групу комунікацій «точка-точка» і до яких класів вони відносяться?

− Операції якої групи реалізуються методами класу `Intracomm`? Назвіть їхню основну властивість.

− Для чого потрібні методи класу `Group`, які типи дій ними підтримуються?

− Якими класами підтримуються дії з віртуальними топологіями процесів? Назвіть декілька дій, які забезпечуються методами цих класів?

5 ЗАСТОСУВАННЯ МЕТОДІВ ПАКЕТУ MPJ EXPRESS ДЛЯ СТВОРЕННЯ ПАРАЛЕЛЬНИХ ПРОГРАМ

В цьому розділі будуть докладно розглянуті питання синтаксису методів пакету, кількість і типи параметрів, які до них передаються і повертаються ними. Будуть наведені приклади використання методів у простих паралельних програмах. Тут слід зауважити, що основним засобом створення паралельних програм у мові Java слугують потоки, а не процеси, як це частіш за все властиво для інших мов програмування. Однак потоки Java, будучи запущені на різних

процесорах в кластерній системі, виглядають майже як процеси, та й засоби бібліотек MPI для інших мов програмування використовують процеси, тому *у подальшому ми будемо користуватись мабуть не зовсім вірним, але, як нам здається, більш придатним в даному випадку терміном «**процес**», щоб не підкреслювати розбіжності в специфічних властивостях мов*.

Ну і перед тим як переходити до безпосереднього розглядання методів класів MPJ Express і їхнього використання в паралельних програмах, для кращого розуміння подальшого матеріалу, слід сказати декілька слів про основні поняття і вимоги стандарту MPI.

5.1 Поняття груп, комунікаторів, ідентифікаторів процесів і повідомлень, механізми обробки помилок у пакеті MPJ Express.

Як вже вказувалось API пакету у повному обсязі підтримує хоча б мінімальні вимоги стандарту MPI. Одним з основних понять програми за технологією інтерфейсу передачі повідомлень є ***група процесів***. Якщо задача потребує локалізації взаємодії паралельних процесів програми, то MPI надає можливість створювати групи процесів. Склад утворених груп довільний і визначається тільки структурою застосованого паралельного алгоритму і топологією зв'язків між процесами (але не процесорами). Групи можуть повністю збігатися, входити одна до одної, не перетинатися або перетинатися частково.

На базі існуючих груп створюються структури на ім'я ***комунікатор***. Це локалізоване окреме середовище для спілкування процесів між собою. Процеси можуть взаємодіяти між собою, тобто відправляти і отримувати повідомлення, лише всередині такого комунікатора, повідомлення, відправлені в різних комунікаторах, не перетинаються й не заважають одне одному. Стандарт MPI визначає тип комунікатора для мов Fortran – INTEGER і C – спеціальний, наперед визначений тип MPI_Comm. В пакеті мовою Java комунікатори є об'єктами класу `Comm` і відповідно сумісні за типом з об'єктами його похідних класів.

Комунікатори поділяються на два класи: ***інтракомунікатори*** – об'єкти похідного класу `Intracomm` для операцій всередині окремої групи процесів і

інтеркомунікатори – об'єкти субкласу `Intercomm` для здійснення операцій між двох груп процесів.

При запуску паралельної програми на виконання в системі, завжди для всіх породжених в ній процесів назначається глобальний всеосяжній комунікатор у межах якого вони працюють і здійснюють взаємодії. За стандартом такий комунікатор має визначене ім'я `MPI_COMM_WORLD`. В програмі мовою Java такий глобальний комуникатор є статичним полем класу MPI, належить до типа Intracomm і має ідентифікатор `COMM_WORLD`. Звернення до нього виконується за допомогою складового ідентифікатора `MPI.COMM_WORLD`. Цей комунікатор існує завжди, він призначений для взаємодії всіх запущених при старті MPI-програми процесів. Пізніше у програмі за потребою можуть створюватись і знищуватися додаткові комунікатори. Крім цього комуніатора, при запуску програми системою ініціюються ще два спеціальних комунікатора (статичних полей класу MPI). По-перше це комунікатор `MPI.COMM_SELF`, який призначений для утримання лише одного – поточного процесу. По-друге це пустий комунікатор у якому не міститься і не може міститись жодного процесу, комунікатор має ім'я `MPI.COMM_NULL`. Мабуть необхідно ще раз нагадати, що всі взаємодії процесів протікають у рамках певного комунікатора, повідомлення, передані в різних комунікаторах, ніяк не заважають одне одному.

Кожен процес MPI-програми в групі, до якої він входить, отримує унікальний атрибут, ідентифікатор або ***ранг*** − ціле невід'ємне число, яке необхідне для розпізнавання процесів і просто має смисл номеру процесу в цієї групі. Найчастіше ранги процесів в групі упорядковані, але це не є обов'язковим, при створенні групи до неї можуть бути додані процеси з неупорядкованими рангами з інших груп. Але потім ранги для зручності можна упорядкувати. За допомогою цього атрибута відбувається значна частина взаємодії процесів між собою. Повністю зрозуміло, що в тому самому комунікаторі всі процеси повинні мати різні номери. Однак, процес може одночасно входити до різних комунікаторів, і його номер в одному комунікаторі може відрізнятися від номера в іншому. Звідси зрозуміла наявність і обов'язковість двох основних атрибутів проце-

су при зверненні до нього: комунікатора і рангу процесу у ньому − ця пара повністю описує процес у системі. Кожна група (комунікатор) має розмір, який визначається кількістю процесів, що входять до неї. Ранги процесів нумеруються починаючи з нуля, тобто якщо група містить N процесів, то ранг будь-якого процесу в даній групі мусить лежати у межах від 0 до N - 1.

Як вже зазначалось, основним способом спілкування процесів між собою є явне пересилання повідомлень від одного до іншого. ***Повідомлення*** − це набір даних деякого визначеного у системі типа. Кожне повідомлення має кілька атрибутів, зокрема, ранг процесу-відправника, ранг процесу-одержувача, ***ідентифікатор повідомлення*** та інші. Одним з важливих атрибутів повідомлення є його ідентифікатор, котрий ще називають ***тег повідомлення***. Тег повідомлення необхідний в тім випадку коли процесу-одержувачу повідомлення необхідно якимось чином розрізнити два окремих повідомлення, що надійшли до нього від одного й того ж самого процесу-відправника. Тег повідомлення, як і ранг процесу, є цілим невід'ємним числом, значення якого лежить у діапазоні від 0 до `MPI.TAG_UB`. Максимальне значення цього поля класу MPI визначається константою `Integer.MAX_VALUE` стандартної бібліотеки мови Java. Але при роботі в системі з `native`-драйверами, значення цього поля визначається відповідною спорідненою бібліотекою MPI і не може бути менш ніж 32767.

Для спрощення роботи з атрибутами повідомлень вони *зберігаються в полях спеціального об'єкта* типа `Status`, що повертається практично всіма методами отримання повідомлень. Об'єкт типа `Status` дає можливість отримати доступ до значень атрибутів одержаних повідомлень.

В класі MPI пакету визначені два статичних поля для опису процесу-відправника з будь-яким рангом – `MPI.ANY_SOURCE` і повідомлення, що має будь-який тег – `MPI.ANY_TAG`. Стандартом MPI значення цих полів-констант не визначається і у бібліотеках різних виробників вони можуть бути різними, тому звернення до них може *виконуватися лише за їхніми ідентифікаторами*.

Стандарт передбачає, що бібліотечні функції повинні повідомляти систему про успішність виконання необхідної операції і безпомилкове завершення

функції, або повертали номер помилки. У мові Fortran вимоги стандарту забезпечуються за рахунок змини функцією значення одного з параметрів, які передаються до функції, за звичаєм це останній параметр набору. У бібліотеках мовою C номер помилки передається у систему завдяки параметру, що повертається з функції. Методи у мові Java в принципі можуть забезпечити будь-який з цих механізмів оповіщення про виникнення помилки. Але розробники пакету класів пішли традиційним для Java шляхом. Практично всі методи класів у разі виникнення помилкової ситуації вкидають виняток, який належить до класу MPIException. Таким чином, всі виняткові ситуації в програмі обробляються за допомогою блоків try/catch, як це прийнято у Java. За бажанням програміста, цей клас може бути розширений з метою більш якісного оповіщення про виниклі помилки програми.

5.1.1 Контрольні запитання та завдання для самостійної роботи

– Що таке група процесів, навіщо таке поняття включено у стандарт MPI, які їхні основні властивості?

– Що таке комунікатор процесів, навіщо він потрібен, чим ця структура відрізняється від групи?

– Комунікатори якого типа Ви знаєте? Чим відрізняється спосіб застосування комунікаторів різних класів?

– Назвіть і охарактеризуйте комунікатори, які створюються в системі при запуску MPI-програми.

– Розкажіть, що таке ранг процесу, для чого він призначений, опишіть його основні властивості.

– Дайте визначення повідомлення, розкажіть, що таке тег повідомлення, для чого він призначений, які значення він може приймати.

– У якій структурі зберігаються атрибути повідомлення? Що це за константи `MPI.ANY_TAG` і `MPI.ANY_SOURCE`, як них застосовувати у програмі?

– Який механізм служить для оповіщення про виникнення помилки в працюючій програмі?

5.2 Методи MPI Express загального призначення

Цій розділ посібника призначений для ознайомлення читачів із загальними методами класу MPI, які напряму не пов'язані з пересиланням даних. Більшість методів, які будуть розглянуті зараз, є необхідними практично в кожній паралельній програмі, що працює із застосуванням пакета класів бібліотеки MPJ Express.

Загальна схема MPI-програми мовою Java може виглядати приблизно таким чином:

```
import mpi.*;
class ClassName
  {
// Оголошення полей класу
// Конструктори класу
    public static void main(String[] args) throws MPIException
      {
// Огодошення змінних, деякі попередні дії програми
        MPI.Init(args);
// Паралельна частина програми
        MPI.Finalize();
// Заключні дії програми
      }
// Інші методи, якщо такі є
  }
```

Першим і абсолютно обов'язковим для застосування у програмі, методом класу MPI є метод, який оголошується наступним чином:

```
public static String[] Init(String[] args) throws MPIException
```

Метод є аналогом процедури `MPI_INIT` стандартної бібліотеки MPI і також як той служить для ініціалізації всіх процесів, що складають паралельну частину програми. *Всі інші методи класу* MPI (*окрім лише* `Initialized()`) *можна викликати тільки після того як буде зроблений виклик методу* `Init`. Ініціалізація паралельної частини для кожного застосування повинна виконува-

тися тільки один раз. До методу `Init` передається посилання на масив `args` параметрів типа `String`. Це той самий масив, який передається системою до методу `main`. Цей параметр є потрібним для того, щоб програма могла одержати й передати до паралельних процесів деякі параметри, які необхідні для їхнього запуску. В свою чергу метод також повертає посилання на масив типа `String`, в якому містяться лише параметри командного рядка, який передається у програму користувачем.

Наступним, теж-таки обов'язковим, призначеним для керування процесами методом, є метод з оголошенням:

```
public static void Finalize() throws MPIException
```

Цей метод також є відповідним аналогом процедури `MPI_FINALIZE` стандартної бібліотеки MPI і здійснює деініціалізацію і завершення одного процесу паралельної частини застосування, який його виклікав. Метод `Finalize` неявно є колективним методом відносно `COMM_WORLD`, тобто він обов'язково повинен бути викликаним всіма процесами в комунікаторі. Але при цьому не потребується виконання повернення з нього всіма процесами. Достатньо що б, як мінімум, повернення виконував процес з рангом 0 в глобальному комунікаторі. Це надає можливість користувачам дізнатися про завершення обчислень в системі.

Після виконання повернення з методу, окрім метода `Initialized` не можна звертатись до жодного методу, навіть виконувати повторний виклик методу `Init`. Перед викликом `Finalize`, кожен процес повинен завершити всі з'єднання, що знаходяться у стані очікування, які він ініціював раніш. Це означає, що після повернення з методу процес вже не може з'єднуватись і брати участь в обміні повідомленнями з іншими процесами, але він може продовжувати виконувати локальні обчислення або дійсно завершитись.

Ілюстрацію застосування цих двох методів, можна зробити на прикладі найпростішої MPI-програми мовою Java (Ex_01). Ця програма має такий вигляд:

```
import mpi.*;
class Ex_01 {
  public static void main(String[] args) throws MPIException {
    System.out.println("До виклику MPI.Init");
    MPI.Init (args);
    System.out.println("Паралельна секція");
    MPI.Finalize ();
    System.out.println("Після MPI.Finalize");
  }
}
```

Те, як будуть друкуватись рядки «До виклику MPI.Init» й «Після MPI.Finalize», одним виділеним процесом, або всіма процесами застосування, що запущені у системі, повністю залежить від того, як реалізована бібліотека MPI. Натомість рядок «Паралельна секція» *завжди повинен бути надрукованим всіма процесами*. Але в цьому разі, *порядок виводу рядків з різних процесів може бути довільним*.

Іноді виникає необхідність з'ясувати виконуються зараз дії у ініційованої частині програми або ні. Для цього до класу MPI доданий метод, який є аналогом процедури `MPI_INITIALIZED`.

```
public static boolean Initialized() throws MPIException
```

Метод повертає значення `true`, якщо він викликається з паралельної частини застосування, і значення `false` – у протилежному випадку. Це єдиний метод MPI, який можна викликати ще до виклику `Init` і після виконання методу `Finalize`.

Є ще декілька методів загального призначення, які вже належать до класу `Comm`, а не MPI. Так у класі `Comm` оголошений метод `Abort` (аналог методу `MPI_ABORT`). Як говорилось вище, будь-який процес перед викликом метода завершення `Finalize` повинен переконатися, що всі взаємодії без блокування, яки знаходились у стані очікування вже завершені, а перед завершенням останнього процесу всі синхронні відправлення, що очікують відповіді, повинні бути зіставленими операціями прийому і навпаки. Але при виклику методу `Abort` такі

вимоги не мають сенсу тому, що метод, будучи викликаним хоча б одним процесом комунікатора, *негайно викликає аварійне завершення всіх процесів* і припинення паралельної програми. Проте метод намагається найкращим способом перервати виконання всіх задач в групі глобального комунікатора.

Метод має наступне оголошення:

```
public void Abort(int errorcode) throws MPIException,
```

параметр `errorcode` – це код помилки для повернення в середовище виконання. Хоча метод і оголошений, але, нажаль, в описуваній версії класів пакету MPJ Express він не виконує ні яких дій і застосовується лише з метою сумісності програми з іншими бібліотеками MPI, або в надії на реалізацію у наступних версіях пакету. У сьогодення при необхідності термінового переривання програми доводиться прибігати до різних хитрощів, або просто звертатись до методу стандартної бібліотеки мови Java – `System.exit(errorCode)`, що може призвести до небажаних наслідків.

Два наступні методи також належать до класу `Comm` і дають можливість отримати деяких параметри комунікаторів і процесів в них.

Перший з оголошенням як:

```
public int Size() throws MPIException
```

є аналогом `MPI_COMM_SIZE` і повертає число паралельних процесів у об'єкті комунікатора якому було надіслано відповідне повідомлення.

І другий − аналог `MPI_COMM_RANK`, з наступнім оголошенням:

```
public int Rank() throws MPIException
```

повертає ранг (номер) процесу в комунікаторі. Якщо метод `Size` для того ж комунікатора раніш повернув значення `size`, то значення, яке повертає метод `Rank` лежить у діапазоні від 0 до `size-1`.

Наступний приклад (Ex_02) демонструє, як кожен запущений процес друкує свій унікальний номер у комунікаторі `COMM_WORLD` і число процесів у даному комунікаторі.

```
import mpi.*;
class Ex_02 {
  public static void main(String[] args) throws MPIException{
    MPI.Init(args);
    int size = MPI.COMM_WORLD.Size();
    int rank = MPI.COMM_WORLD.Rank();
    System.out.printf("Процес %d-й з %d", rank, size);
    MPI.Finalize();
  }
}
```

Рядок, що відповідає виклику методу друку `printf`, буде виведений стільки разів, скільки процесів були породжені у момент створення паралельної частини запущеної програми. Порядок появи рядків не є заздалегідь визначеним і, власне кажучи, може бути довільним. Гарантується тільки те, що вміст окремих рядків не буде перемішаним один з одним.

Статичний метод класу MPI

```
public static double Wtime() throws MPIException
```

до процесу, що його викликав, повертає дійсне число подвійної точності, яке являє собою астрономічний час у секундах, який відраховується з деякого моменту в минулому (так званий системний час). Якщо якусь ділянку програми на початку й наприкінці оточити викликами даного методу, то різниця повернутих ним значень, визначить час роботи ділянки. Гарантується, що момент часу, який використовується як початкова точка відліку, не буде змінений за час існування процесу. Метод є аналогом функції `MPI_WTIME` зі стандартної бібліотеці MPI.

Є доволі незручним те, що метод повертає час у секундах, тому він може застосовуватись лише для вимірювання тривалих проміжків часу. Для вимірю-

вання швидкоплинучих процесів краще застосовувати стандартний метод класу `System` мови Java – `carrentTimeMillis()`.

Природним є той факт, що таймери різних процесорів, на яких запускається паралельне застосування, можуть бути не синхронізовані й видавати різні значення. Для з'ясування наявності такої розсинхронізації у стандартній бібліотеці MPI передбачений параметр `MPI_WTIME_IS_GLOBAL` значення якого дорівнює 1 − коли процесори синхронізовані та 0 − якщо ні. Нажаль, у *реалізації бібліотеки*, яка застосовується у лабораторних роботах, *така можливість відсутня*.

Існує ще один метод класу MPI, який дає часову характеристику процесора – це аналог функції `MPI_WTICK`

```
public static double Wtick() throws MPIException
```

він повертає тривалість такту таймера у секундах для процесора, на якому працює процес, який викликав цей метод.

У класі MPI є один спеціальний метод, за допомогою якого можна визначити, на які саме фізичні процесори системою були сплановані процеси MPI-застосування. Метод є аналогом процедури `MPI_GET_PROCESSOR_NAME` і оголошується як

```
public static String Get_processor_name() throws MPIException
```

Метод повертає ім'я вузла, на якому запущений процес, що його викликав.

У наступній програмі (Ex_03) для кожного процесу визначаються дві характеристики системного таймера: тривалість його такту і час, необхідний для проведення одного вимірювання часу. Для отримання більш точного значення, виконується `nTimes` вимірів з наступним усередненням результатів. Кількість `nTimes` повторень циклу вимірювання часу передається у програму як параметр командного рядка системи. Відсутність цього параметра командного рядка ви-

кликає вкидання винятку типа ArrayIndexOutOfBoundsException. У даному прикладі також показане використання методу Get_processor_name.

```
import mpi.*;
class Ex_03 {
  private static int rank;
  private int nTimes;
  private String name;
  private double tick;
  private final String MESSTR =
    " %nНа процесорі%8s процес%2d: tick = %f с," +
    " час роботы = %f с";
  public Ex_03(String[] args) {
    String str[] = MPI.Init(args);
    rank = MPI.COMM_WORLD.Rank();
    nTimes = Integer.parseInt(str[0]);
    name = MPI.Get_processor_name();
    tick = MPI.Wtick();
  }
  private void run() {
    double time, time_start, time_finish;
    time_start = MPI.Wtime();
    for(int i = 1; i < nTimes; i++) time_finish = MPI.Wtime();
    time_finish = MPI.Wtime();
    time = (time_finish - time_start);
    System.out.printf(MESSTR, name, rank, tick, time / nTimes);
  }
  public static void main(String[] args) throws MPIException {
    String ERRSTR =
      " %nНе вистачає параметра для методу main." +
      "%nСинтаксис команди запуску програми:%n" +
      " %n\t\tmpjrun.sh -np <К-ТЬ ПРОЦ.> Ex_03 <К-ТЬ ЦИКЛІВ>";
    try {
      Ex_03 ex = new Ex_03(args);
      ex.run();
    }
    catch(ArrayIndexOutOfBoundsException exc) {
      if(rank == 0) System.err.printf(ERRSTR);
    }
    finally { MPI.Finalize(); }
    if(rank == 0) System.out.printf("%n %n");
  }
```

```
}
```

5.2.1 Контрольні запитання та завдання для самостійної роботи

– Опишіть приблизний вигляд схеми MPI-програми мовою Java.

– Розкажіть про статичний метод класу MPI – метод `Init`. Які його особливості Ви можете назвати? Які параметри йому передаються і які повертаються?

– Чи можна у MPI-прграмі обійтись без застосування метода `MPI.Finalize`? Які особливості має цей метод при використанні?

– Для чого призначений метод `MPI.Initialized`, чи можна його застосовувати до виклику метода `Init` і після використання метода `Finalize`?

– Чим відрізняється застосування методу `MPI.Abort` від застосування метода `Finalize` у стандарті MPI, як реалізований метод `Abort` в описуваній бібліотеці MPJ Express?

– Як визнати розмір комунікатора і ранги процесів, які до нього належать, а також визначити ім'я вузла мережі, на якому запущений і виконується процес?

– Чим відрізняється застосування методів `MPI.Wtime` і `MPI.Wtick`, для вимірювання яких параметрів системи вони призначені? Який (які) стандартний метод бібліотеки класів мови Java можна застосовувати замість метода `Wtime`?

– Можна чи ні у MPI-програмі з використанням класів пакету MPJ Express, яка запущена на і працює на кластері (кластер робочих станцій теж кластер), визначити синхронізовані чи ні таймери різних процесів такої системи? І якщо ні, але дуже хочеться, то як це можна зробити?

– Відкомпілюйте і запустить програму Ex_01. Визначте, скільки процесів виконують текст програми до виклику методу `MPI.Init` і після виклику методу `MPI.Finalize` у описуваній системі.

– Напишіть програму HelloAuthor.java, у який кореневий процес з рангом 4 приймає з консолі Ваше ім'я, розсилає його іншим процесам глобального комунікатора і всі процеси виводять на консоль привітання для Вас з зазначенням власного рангу. Програму запустить з використанням 10 процесів.

− Змінить програму так, щоб останній процес комунікатора одержував Ваше ім’я з командного рядка системи. Інші дії програми відповідають попередньому завданню. Тричі запустить програму з різною кількістю процесів (наприклад, 3, 8 і 15).

− Ще раз змінить програму так, щоб вона з командного рядка системи одержувала не тільки Ваше ім’я, а й ранг кореневого процесу.

5.3 Операції комунікації типа «точка-точка» між окремими процесами

Як вже відмічалось, взаємодія між процесами при програмуванні з використанням бібліотеки MPI здійснюється за допомогою явного пересилання повідомлень. Практично всі програми, які написані з використанням комунікаційної технології MPI, повинні містити не тільки засоби для породження й завершення паралельних процесів, але й для здійснення взаємодії запущених процесів між собою.

Всі методи передачі повідомлень в MPI поділяються на дві групи. До однієї з них входять методи, призначені для взаємодії тільки двох процесів програми. Такі операції називаються ***індивідуальними*** або операціями ***типа точка-точка***. Методи іншої групи припускають, що до взаємодії під час операції повинні бути залучені всі процеси деякого комунікатора. Такі операції називаються ***колективними*** (див. нижче).

Оскільки взаємодія процесів за допомогою операцій передачі повідомлень може виконуватись винятково лише у межах того або іншого комунікатора, то відповідні методи, які надсилають і приймають повідомлення, належать до об’єктів, що мають тип `Comm`, або його похідних – `Intercomm`, `Intracomm`, або `Cartcomm` та `Graphcomm`.

Опис методів обміну повідомленнями між процесами природно почати з обговорення операцій типа точка-точка. У таких взаємодіях беруть участь лише два об’єкти-процеси, причому один процес виступає як відправник повідомлення, а інший − як одержувач. Процес-відправник, викликаючи один з методів передачі даних, *повинен явним чином вказати номер (ранг) процесу-одержувача* у

тому ж комунікаторі, до якого належить і він. У відповідь на ці дії відправника, процес-одержувач *повинен викликати один з методів прийому повідомлення*, указавши ранг процесу від якого очікувалось повідомлення і який належить до того ж об'єкта комунікатора. При цьому, в деяких випадках, одержувач може не знати точний номер процесу-відправника в даному комунікаторі.

Всі методи даної групи пересилань, у свою чергу, так само поділяються на два класи:

– методи ***з блокуванням процесів*** (***з синхронізацією***)

– методи ***без блокування*** (***асинхронні***).

Методи обміну з блокуванням припиняють роботу процесу і залишаються незавершеними до виконання деякої умови, а повернення з асинхронних методів відбувається негайно після ініціалізації відповідної комунікаційної операції. Сам принцип взаємодії процесів у програмі з використанням методів з блокуванням потребує особливої додаткової обережності, тому що неакуратне поводження з ними може призвести до виникнення так званої ***тупикової ситуації***. Використання асинхронних операцій ніколи не призводить до тупикових ситуацій, однак їхне застосування може викликати втрату даних, тому програмісту потрібно більш акуратно ставитися до використання масивів даних, які пересилаються такими операціями.

У стандартному режимі операції обміну типа точка-точка виконуються в три етапи:

Спочатку на передавальному боці у буфері відправника формується пакет повідомлення, до якого крім інформації, що потрібна для обчислювального процесу, ще додатково впаковуються адреса відправника (`source`), адреса одержувача (`dest`) та ідентифікатор повідомлення (`tag`). Потім, в залежності від версії бібліотеки MPI, цей пакет системними засобами або передається безпосередньо від буфера відправника до буфера приймача повідомлення, або поміщається у деякий системний буфер, і на цьому функція посилки повідомлення завершується.

На останньому етапі приймаючий процес витягає повідомлення із власно-

го або системного буфера тоді, коли в нього з'являється потреба в даних. Змістовна частина повідомлення поміщається в адресний простір приймаючого процесу (параметр `buf`), а службова в поля об'єкта типа `Status`.

В загальному випадку об'єкт `status`, у який поміщається адресна частина прийнятого повідомлення, має три поля:

– ідентифікатор комунікатора (`comm`), оскільки кожний процес може одночасно входити в кілька областей зв'язку;
– номер відправника в цій області зв'язку (`source`);
– ідентифікатора повідомлення (`tag`), що використовується для взаємної прив'язки конкретної пари операцій посилки й прийому повідомлень.

Але у пакеті MPJ Express цих цілей розробники задіяли лише два поля – `source` і `tag`.

5.3.1 Передача/прийом повідомлень з блокуванням

Для здійснення відправлення повідомлення з блокуванням в класі `Comm` оголошений метод

```
public void Send(Object buf, int offset, int count,
                 Datatype datatype, int dest, int tag)
                                        throws MPIException
```

він виконує пересилання частини буферу даних `buf`, який містить елементи типа `Datatype`. Пересилання завжди починається з позиції `offset` у буфері, а кількість елементів повідомлення дорівнює `count`. Повідомлення має ідентифікатор `tag` та призначено воно процесу з номером `dest`. Всі елементи повідомлення, що пересилається, повинні розташовуватись підряд у буфері `buf`. Операція починається незалежно від того, був ініційований відповідний метод одержання повідомлення чи ні, але завершення її можливе тільки за умови початку прийому повідомлення. Повідомлення, зазвичай, копіюється з буфера процесу-відправника безпосередньо до буфера прийому процесу-одержувача, однак, існують такі реалізації бібліотеки MPI, у яких передбачено попереднє збереження

даних у деякому системному буфері. Параметр `count` може приймати значення від 0 до `buf.length`. В загальному випадку процесу дозволяється передавати повідомлення самому собі, однак, це є дуже небезпечним тому, що може призвести до виникнення тупикової ситуації.

Для позначення параметра `datatype` *обов'язково слід користуватись ідентифікаторами* кількох загальнодоступних статичних полів (`public static`) типа `Datatype`, які оголошені у класі MPI пакету. У табл. 5.1, у порівнянні з примітивними типами мови Java і типами зі стандарту інших бібліотек MPI, наведені найменування типів з пакету MPJ Express, які використовуються з метою позначення типа даних.

Таблиця 5.1 − Типи даних у пакеті MPJ Express у порівнянні з примітивними і стандартними

Типи у пакеті MPJ	Тип даних у мові Java	Звичайні типи в MPI
MPI.BYTE	byte	MPI_BYTE
MPI.CHAR	char	MPI_CHARACTER
MPI.SHORT	short	
MPI.BOOLEAN	boolean	MPI_LOGICAL
MPI.INT	int	MPI_INTEGER
MPI.LONG	long	
MPI.FLOAT	float	MPI_REAL
MPI.	double	MPI_DOUBLE_PRECISION
MPI.	Object	
MPI.		
MPI.		MPI_INTEGER2
MPI.LONG2		
MPI.FLOAT2		MPI_REAL2
MPI.DOUBLE2		
MPI.PACKED		MPI_PACKED

Стандарт MPI щодо типів даних, яких повинні додержуються всі розробники таких бібліотек, вимагає щоб в бібліотеці MPI обов'язково забезпечуватись таки типи даних які відповідають всім специфічним типам базової мови, яка використовується разом з MPI. Як видно з наведеної таблиці, розробники пакету MPJ Express цілком задовольнили вимоги стандарту додавши такі типи як `SHORT` і `LONG`, що відповідають примітивним типам `short` і `long` мови Java.

Добавлений також тип OBJECT, що відповідає типу Object.

В класі Datatype оголошено ще кілька типів, які не мають відповідних прототипів у мові Java. По-перше це тип, який визначається як MPI.PACKED. Цей тип мають, так звані, ***впаковані дані*** і визначає спеціальним образом зібрані дані (іноді різних типів) і вміщені у буфер пересилання повідомлень. Цілком зрозуміло, що у Java такого типа не існує.

По-друге це сімейство типів з суфіксом 2, наприклад, INT2 або FLOAT2 та інші. У стандарті MPI подібні типи являють собою двійки чисел відповідних типів, мають обмежене використання і призначені для оголошення і використання типів у методах редукування даних при знаходженні мінімальних (максимальних) значень і їхніх індексів у масивах даних. У пакеті MPJ Express таки дані просто є впакованими даними з пари відповідних чисел і являють собою масив даних базового типа, які об'єднані у окремий об'єкт добре пристосований для операцій з обміну даними.

При пересиланні повідомлень до процесу, який у даний проміжок часу не існує, стандарт MPI передбачає використання спеціального рангу – PROC_NULL, який призначається такому процесу. Очікується, що звертання до процесу с таким ідентифікатором не призводить до виникнення помилок у програми і операція з таким процесом просто негайно завершується. Однак методи класів пакету MPJ Express в цьому випадку генерують виняток типа MPIException, який обов'язково повинний бути оброблений відповідним чином.

Нижче наведений приклад фрагмента коду програми, для пересилання повідомлення від довільного процесу до наступного, номер якого на одиницю більше рангу відправника:

```
...
int size = MPI.COMM_WORLD.Size();
int rank = MPI.COMM_WORLD.Rank();
int next = (rank == size - 1) ? MPI.PROC_NULL : rank + 1;
try {
  MPI.COMM_WORLD.Send(buf, 0, 1, MPI.FLOAT, next, 5); }
catch (MPIException e) {...}
```

```
...
```

У цьому випадку, процес із останнім номером у комунікаторі не здійснює ніякого реального пересилання даних, натомість вкидається виняток, який обробляється у операторі `catch`, це не призводить до переривання програми і вона може продовжити виконатись далі.

Повідомлення, які виконуються з блокуванням, гарантують коректність повторного використання всіх параметрів після повернення з методу. Це означає, що після повернення з `Send` можна використовувати будь-які присутні у виклику даного методу змінні без побоювання зіпсувати раніше передане повідомлення. Вибір способу здійснення цієї гарантії: копіювання в проміжний буфер або безпосередня передача даних процесу `dest`, залишається за розроблювачами конкретної реалізації MPI.

Окремо слід зазначити, що повернення з методу `Send` не означає того, що повідомлення вже отримане процесом `dest`, воно навіть не означає того, що повідомлення покинуло процесор, на якому виконується процес, котрий виконав виклик цього методу. Як відмічалось вище, надається тільки гарантія щодо безпечної зміни тих змінних, які були використані у виклику даного методу. Подібна невизначеність далеко не завжди влаштовує користувачів. Для розширення можливостей передачі повідомлень і гарантування отримання них одержувачем, в MPI додані три додаткових методи. Всі параметри в цих методах такі ж самі, як і в `Send`, однак у кожного з методів є своя особливість.

Першою модифікацією методу є метод

```
public void Bsend(Object buf, int offset, int count,
                  Datatype datatype, int dest, int tag)
                                        throws MPIException
```

Це метод *передачі повідомлення з буферуванням*. Виконання даного методу ніяк не залежить від того прийшов або ні відгук від процесу якому призначалося повідомлення про його одержання. Якщо процес-одержувач, на момент

здійснення відправлення, не ініціював прийом повідомлення від процесу-відправника, то повідомлення переноситься в спеціальний буфер для подальшого використання одержувачем, а потім відбувається негайне повернення з методу. Проте, метод може вкинути виняток якщо місця для буфера виявиться замало. Про виділення масиву для буферування повідомлення повинен піклуватися сам користувач. Резервування спеціального масиву для процесу-відправника, який буде використовуватись як буфер, виконується за допомогою спеціального методу класу MPI:

```
public static void Buffer_attach(java.nio.ByteBuffer userBuffer)
                                                  throws MPIException
```

Метод – аналог процедури `MPI_BUFFER_ATTACH` і виділяє для використання масив `userBuffer` типа `ByteBuffer`. Такий тип вибраний з метою одержання можливості поміщати у буфер дані будь-якого типа. Розмір масиву, який потрібно виділити для буферування, можна довідатися за допомогою методу об'єкта класу `Comm` – аналога процедури `MPI_PACK_SIZE`:

```
public int Pack_size(int incount, Datatype datatype)
                                                  throws MPIException
```

і повинен перевищувати обчислений розмір як мінімум на величину, обумовлену значенням статичного поля `BSEND_OVERHEAD` класу MPI.

У кожному процесі може бути тільки один такий буфер. Масив, асоційований з буфером, *ні в якому разі не слід використовувати в програмі для інших цілей*.

Коли буфер стає вже непотрібним, то для одержання можливості використання його в інших цілях необхідним є вивільнення буфера, яке виконується за допомогою методу класу MPI (аналога процедури `MPI_BUFFER_DETACH`)

```
public static void Buffer_detach() throws MPIException
```

Процес, що викликав метод вивільнення буфера, блокується доти, поки всі повідомлення не будуть вибрані і не підуть із даного буфера.

За звичаєм в системах MPI заздалегідь виділяється деякий обсяг пам'яті для буферування повідомлень, які пересилаються у системі. Однак, щоб не покладатися на особливості конкретної реалізації, рекомендується явно виділяти в програмі достатній буфер для всіх пересилань з буферуванням.

У наступному прикладі (Ex_04) демонструється використання передачі повідомлення з буферуванням. Для буферування виділяється масив `buf`, після завершення пересилання він звільняється. Розмір необхідного буфера визначається розміром повідомлення в одне ціле число плюс значення поля `BSEND_OVERHEAD`.

```
import java.nio.*;
import mpi.*;
class Ex_04 {
  private int rank, sendBuf[] = {0};
  private ByteBuffer buf;
  public Ex_04(String[] args) {
    MPI.Init(args);
    rank = MPI.COMM_WORLD.Rank();
    sendBuf[0] = rank;
    int bufSize = MPI.COMM_WORLD.Pack_size(1, MPI.INT) +
                  MPI.BSEND_OVERHEAD;
    buf = ByteBuffer.allocate(bufSize);
  }
  private void run() {
    Status status;
    int[] recvBuf = new int[1];
    if(rank == 0) {
      MPI.Buffer_attach(buf);
      MPI.COMM_WORLD.Bsend(sendBuf, 0, 1, MPI.INT, 1, 5);
      MPI.Buffer_detach(); }
    if(rank == 1) {
      status = MPI.COMM_WORLD.Recv(recvBuf, 0, 1, MPI.INT, 0, 5);
      System.out.printf(" \nПроцес%2d одержав%2d від процесу%2d",
                        rank, recvBuf[0], status.source); }
  }
  public static void main(String[] args) throws MPIException {
```

```
      try {
        Ex_04 ex = new Ex_04(args);
        ex.run();
      }
      catch (MPIException e) {
        System.err.printf(" %nПомилковий параметр командного" +
                          " рядка \"-np %s\"%nКількість процесів" +
                          " не може бути меньшою за 2", args[1]);
      }
      finally { MPI.Finalize(); }
      if (rank == 1) System.out.printf("%n %n");
    }
  }
```

Інша модифікація методу (аналог процедури `MPI_SSEND`) − це *метод передачі повідомлення з синхронізацією*. Він має наступне оголошення:

```
public void Ssend(Object buf, int offset, int count,
                  Datatype datatype, int dest, int tag)
                                          throws MPIException
```

Вихід з даного методу відбувається тільки тоді, коли прийом повідомлення від відправника буде ініційований процесом-одержувачем. Таким чином, завершення передачі із синхронізацією говорить не тільки про можливість повторного використання буфера пересилання, але й про гарантоване досягнення процесом-одержувачем точки прийому повідомлення в програмі. Використання передачі повідомлень із синхронізацією може трохи сповільнити виконання програми, але дозволяє уникнути наявності в системі великої кількості не одержаних буферізованих повідомлень.

Ще одна модифікація методу відправлення повідомлення `Send` має ім’я `Rsend`, слугує аналогом процедури `MPI_RSEND` і має наступне оголошення:

```
public void Rsend(Object buf, int offset, int count,
                  Datatype datatype, int dest, int tag)
                                          throws MPIException
```

Цей метод *забезпечує передачу повідомлення за готовністю*. Даним методом можна користуватися тільки в тому випадку, якщо процес-одержувач уже ініціював прийом повідомлення. У противному випадку виклик методу є помилковим і результат його виконання не може бути визначеним. Гарантувати ініціалізацію прийому повідомлення перед викликом методу `Rsend` можна за допомогою операцій, що здійснюють явну або неявну синхронізацію процесів, наприклад, метод `Barrier` класу `Intracomm` (цей метод явної синхронізації буде розглянутий пізніше) або `Ssend`. У багатьох реалізаціях бібліотек метод `Rsend` скорочує протокол взаємодії між відправником й одержувачем, зменшуючи накладні витрати на організацію передачі даних.

Одержання відправлених повідомлень, *у незалежності від методу їхнього відправлення*, виконується за допомогою одного й того ж методу, який має наступне оголошення:

```
public Status Recv(Object buf, int offset, int count,
                   Datatype datatype, int source, int tag)
                                          throws MPIException
```

Метод є аналогом процедури `MPI_RECV` і забезпечує прийом з блокуванням даних до буферу `buf` не більше `count` елементів повідомлення починаючи з позиції `offset`. Повідомлення *повинне мати той самий тип* `datatype`, що і при його відправленні. Обов'язковими атрибутами повідомлення є тег – `tag` і ранг процесу у комунікаторі – `source` від якого воно було відправлене. Ці атрибути можуть приймати наперед визначені значення `MPI.ANY_TAG` і `MPI.ANY_SOURCE`, відповідно. Метод повертає об'єкт класу `Status`, в полях якого міститься інформація про атрибути отриманого повідомлення. Якщо число реально прийнятих елементів менше значення `count`, то гарантується, що в буфері `buf` зміняться тільки елементи, що відповідають елементам прийнятого повідомлення. Якщо кількість елементів у прийнятому повідомленні більше значення `count`, то виникає виняток виходу індексу за межі масиву. Щоб уникнути цього, можна спочатку за допомогою методів `Probe` або `Iprobe` (див. ниж-

че) визначити структуру повідомлення, що надійшло. Якщо потрібно визначити тільки точне число елементів у прийнятому повідомленні, то можна скористатися методом класу `Status` – `Get_count` (див. нижче).

Блокування процесу одержання повідомлення гарантує, що після повернення з методу `Recv` всі елементи повідомлення вже будуть прийняті й розташовані в буфері одержувача – `buf`.

Нижче наведений приклад програми (Ex_05) у якій нульовий процес посилає повідомлення процесу з номером один і чекає від нього відповіді. Якщо програма буде запущена з числом процесів, яке перебільшує два, то реально виконувати пересилання будуть тільки нульовий і перший процеси. Інші процеси, після їхньої ініціалізації методом `Init` у конструкторі класу, не виконуючи жодних корисних дій просто завершаться звернувшись до методу `Finalize` в блоці `finally` методу `main` програми. У противному випадку, якщо програма буде запущена на виконання з одним процесом, у конструкторі класу буде викинутий виняток типа `MPIException`. У програмі можливо виникнення винятків ще двох типів – це `ArrayIndexOutOfBoundsException` і `NumberFormatException`. Перший з них вкидається у разі відсутності у командному рядку системи параметра програми, який задає розмір масивів дійсних чисел для виконання обміну між двома процесами. А другий сповіщає користувача програми про те, що переданий параметр не є числовим.

```
import mpi.*;
import java.util.*;
class Ex_05 {
  private static int rank;
  private static String[] arg;
  private double[] a, b;
  public Ex_05(String[] args) {
    arg = MPI.Init(args);
    rank = MPI.COMM_WORLD.Rank();
    if (MPI.COMM_WORLD.Size() < 2) throw new MPIException();
    int arrSize = Integer.parseInt(arg[0]);
    a = new double[arrSize];
    b = new double[arrSize];
```

```
  Arrays.fill(a, 1.0);
  Arrays.fill(b, 2.0);
}
private void run() {
String prevMess = " %nДо обміну у процесі%2d:%n\ta = %s;" +
                  "%n\tb = %s.",
       postMess = " %nПісля обміну у процесі%2d:%n\ta = %s;" +
                  "%n\tb = %s.";
  if(rank == 0) {
    Arrays.fill(b, 1.0);
    System.out.printf(prevMess, rank, Arrays.toString(a),
                      Arrays.toString(b));
    MPI.COMM_WORLD.Send(b, 0, b.length, MPI.DOUBLE, 1, 5);
    MPI.COMM_WORLD.Recv(a, 0, a.length, MPI.DOUBLE, 1, 5);
    System.out.printf(postMess, rank, Arrays.toString(a),
                      Arrays.toString(b)); }
  if(rank == 1) {
    Arrays.fill(a, 2.0);
    System.out.printf(prevMess, rank, Arrays.toString(a),
                      Arrays.toString(b));
    MPI.COMM_WORLD.Recv(b, 0, b.length, MPI.DOUBLE, 0, 5);
    MPI.COMM_WORLD.Send(a, 0, a.length, MPI.DOUBLE, 0, 5);
    System.out.printf(postMess, rank, Arrays.toString(a),
                      Arrays.toString(b)); }
}
public static void main(String[] args) throws MPIException {
  try {
    Ex_05 ex = new Ex_05(args);
    ex.run();
  }
  catch(MPIException e) {
    System.err.printf(" %nПомилковий параметр командного" +
                      " рядка \"-np %s\"%nКількість процесів" +
                      " не може бути меньшою за 2", args[1]);
  }
  catch(e) {
    if (rank == 0)
      System.err.printf(" %nНе вистачає параметра для" +
                        " методу main.%n" +
                        "Синтаксис команди запуску програми" +
                        ":%n %n\t\tmpjrun.sh -np <К-ТЬ_ПРОЦ" +
                        "> Ex_05 <РОЗМІР_МАСИВІВ>");
  }
```

```
      catch(e) {
        if (rank == 0)
          System.err.printf(" %nПомилка у командному рядку.%n" +
                            "Параметр програми \"%s\" не є" +
                            " числом!", arg[0]);
      }
      finally { MPI.Finalize(); }
      if (rank == 0) System.out.printf("%n %n");
    }
  }
```

У наступному прикладі (Ex_06) кожен процес, що має парний номер (сюди ж віднесений й процес з рангом 0) посилає повідомлення своєму сусідові з номером на одиницю більшим. Додатково поставлена перевірка для процесу з максимальним номером, щоб він не послав повідомлення неіснуючому процесу. Значення, що містяться у масиві b, зміняться тільки у процесах з непарними номерами. Так само, як і у попередньому випадку, відсутність у командному рядку системи параметра, що передається до програми, або зазначення не числового параметра програми, викликає вкидання відповідних винятків. Параметром програми користувач задає розмір цілочисельних масивів для обміну між процесами.

```
import mpi.*;
import java.util.*;
class Ex_06 {
  private int size, a[], b[];
  private static int rank;
  private static String[] st;
  public Ex_06 (String[] args) {
    st = MPI.Init(args);
    size = MPI.COMM_WORLD.Size();
    rank = MPI.COMM_WORLD.Rank();
    int arrLen = Integer.parseInt(st[0]);
    a = new int[arrLen];
    b = new int[arrLen];
    Arrays.fill(a, rank);
    Arrays.fill(b, -1);
  }
```

```
  private void run(String[] args) {
      if(rank % 2 == 0) {
// Повідомлення відправляють лише всі процеси з парними
// номерами, крім останнього парного при непарній кількості
// процесів в комунікаторі
        if(rank != (size - 1))
          MPI.COMM_WORLD.Send(a, 0, a.length, MPI.INT,
                              rank + 1, 5);
      }
      else
        MPI.COMM_WORLD.Recv(b, 0, b.length, MPI.INT, rank - 1, 5);
      System.out.printf(" %nПроцес %d:%n  a = %s;%n  b = %s.",
                        rank, Arrays.toString(a),
                        Arrays.toString(b));
    }
  public static void main(String[] args) throws MPIException {
    try {
      Ex_06 ex = new Ex_06(args);
      ex.run(args);
    }
    catch(ArrayIndexOutOfBoundsException exc) {
      if(rank == 0)
        System.err.printf(" %nНе достатньо параметрів для" +
                    " метода main.%n " +
                    "Синтаксис команди запуску программи:%n" +
                    " %n\t\tmpjrun.sh -np <К-ТЬ_ПРОЦ.> Ex_3" +
                    " <РОЗМІР_МАСИВІВ>");
    }
    catch(NumberFormatException e) {
      if (rank == 0)
        System.err.printf(" %nПомилка у командному рядку.%n" +
                          "Параметр програми \"%s\" не є числом!",
                          st[0]);
    }
    finally { MPI.Finalize(); }
    if (rank == 0) System.out.printf("%n %n");
  }
}
```

У методі `Recv`, коли не відомі ранг процесу, що надіслав повідомлення, і/або тег повідомлення, при прийомі повідомлення замість аргументів `source` й

tag можна використовувати значення статичних полів класу MPI, які мають наступні ідентифікатори:

– ANY_SOURCE – це ознака того, що можна приймати повідомлення від будь-якого процесу;
– ANY_TAG – ознака того, що підходить повідомлення з будь-яким тегом.

При одночасному використанні цих двох констант буде прийняте повідомлення з будь-яким тегом від будь-якого процесу.

Реальні атрибути прийнятого повідомлення завжди можна довідатися визначивши них з відповідних полів об'єкта типа Status. У полях source і tag цього об'єкта зберігаються номер процесу-відправника повідомлення і ідентифікатор повідомлення, відповідно.

Слід мати на увазі деяку несиметричність операцій пересилання й прийому повідомлень. За допомогою ідентифікатора ANY_SOURCE можна прийняти повідомлення від будь-якого процесу. Однак у *випадку пересилання даних застосування цієї константи не можливо і треба явно вказувати номер приймаючого процесу*.

У стандарті окремо обговорюється, що *коли один процес послідовно посилає два повідомлення до іншого процесу* і вони обидва відповідають тому ж самому виклику методу Recv, то *першим буде одержано повідомлення, що було відправлено раніше*. Разом з тим, якщо два *повідомлення були одночасно відправлені різними процесами, то порядок їхнього одержання приймаючим процесом заздалегідь не визначений*, в цьому разі вже може знадобитися проведення аналізу полів об'єкта типа Status, який повертається методом.

В класі Status, крім вказаних полів-членів ще оголошений метод, який, до речі, вже згадувався – це аналог процедури MPI_GET_COUNT

```
public int Get_count (Datatype datatype) throws MPIException
```

Цей метод повертає число елементів типа datatype в повідомленні, яке вже одержано *після звертання до методу* Recv, або в тих повідомленнях, що

будуть прийняти після звертання до методів Probe або Iprobe. Метод Get_count, зокрема, є необхідним для визначення розміру області пам'яті, яка потрібна для зберігання повідомлення, яке буде одержано.

В свою чергу аналог процедури MPI_PROBE – метод класу Comm

```
public Status Probe(int source, int tag) throws MPIException
```

теж повертає об'єкт типа Status і дає можливість одержати в його полях інформації про структуру очікуваного повідомлення із блокуванням, яке має ідентифікатор tag та відправлене від процесу з номером source. Повернення з методу не відбудеться доти, поки повідомлення з підходящим ідентифікатором і номером процесу-відправника не буде доступно для одержання. Слід особливо звернути увагу на те, що метод визначає тільки факт приходу повідомлення, але реально його прийому не здійснює. Якщо після виклику Probe одразу викликається Recv з тими ж самими параметрами, то буде прийняте те ж саме повідомлення, інформація про яке була отримана за допомогою виклику методу Probe.

Наступний приклад програми (Ex_07) демонструє застосування методу Probe для визначення структури повідомлення, що буде отримано. У прикладі процес з рангом 0 чекає повідомлення з однаковим тегом від кожного з двох процесів з номерами 1 й 2. Однак дані, що посилаються цими процесами, мають різний тип. Для того щоб заздалегідь визначити, у яку змінну поміщати повідомлення, що надходить, процес спочатку за допомогою виклику Probe визначає, від кого ж саме надійшло це повідомлення. Наступний безпосередньо після звернення до Probe виклик Recv гарантовано прийме потрібне повідомлення, після чого буде виконаний прийом повідомлення від іншого процесу.

```
import mpi.*;
class Ex_07 {
  private static int rank, root = 0;
  private int[] ibuf;
  private double[] rbuf = new double[1];
  public Ex_07(String args[]) throws MPIException {
    MPI.Init(args);
```

```
  rank = MPI.COMM_WORLD.Rank();
  if (MPI.COMM_WORLD.Size() < 3) throw new MPIException();
  ibuf = new int[1]; ibuf[0] = rank;
  rbuf = new double[1]; rbuf[0] = (double)rank;
}
private void run() {
  String format =
    " %nПроцес %d одержав:%n\t%d від процесу %d;" +
    "%n\t%f від процесу %d.";
  int[] sender = new int[2];
  switch (rank) {
    case 1: {
      MPI.COMM_WORLD.Send(ibuf, 0, 1, MPI.INT, root, 5);
      break; }
    case 2: {
      MPI.COMM_WORLD.Send(rbuf, 0, 1, MPI.DOUBLE, root, 5);
      break; }
    case 0: {
      Status status = MPI.COMM_WORLD.Probe(MPI.ANY_SOURCE, 5);
      if (status.source == 1) {
        sender[0] = status.source; sender[1] = rank + 2;
        MPI.COMM_WORLD.Recv(ibuf, 0, 1, MPI.INT,
                            sender[0], 5);
        MPI.COMM_WORLD.Recv(rbuf, 0, 1, MPI.DOUBLE,
                            sender[1], 5); }
      else {
        sender[1] = status.source; sender[0] = rank + 1;
        MPI.COMM_WORLD.Recv(rbuf, 0, 1, MPI.DOUBLE,
                            sender[1], 5);
        MPI.COMM_WORLD.Recv(ibuf, 0, 1, MPI.INT,
                            sender[0], 5); }
      System.out.printf(format, rank, ibuf[0], sender[0],
                        rbuf[0], sender[1]);
    }
  }
}
public static void main(String args[]) throws MPIException {
  final String ERRSTR =
    " %nПомилковий параметр \"-np %s\" у командному рядку." +
    "%nКількість процесів повинна бути не менша за 3.";
  try {
    Ex_07 ex = new Ex_07(args);
    ex.run(); }
```

```
    catch (MPIException e) {
      if (rank == root) System.err.printf(ERRSTR, args[1]);
/*
 *Тут бажано б було використати щось на кшталт:
 *      MPI.COMM_WORLD.Abort(0x7f);
 *для припинення виконання програми, але ...
 */
    }
    finally { MPI.Finalize(); }
    if (rank == root) System.out.printf("%n %n");
  }
}
```

У наступному прикладі (Ex_08) моделюється послідовний обмін повідомленнями між двома процесами. При цьому проводиться вимірювання часу потрібного на одну ітерацію обміну і визначається залежність часу обміну від довжини повідомлення. Таким чином, визначаються базові характеристики комунікаційної мережі паралельного комп'ютера: ***латентність*** − час затримки повідомлення або час необхідний для передачі повідомлення нульової довжини і ***максимально досяжна пропускна здатність*** комунікаційної мережі − максимальна кількість байтів, яку можна передати по мережі за одиницю часу. Також, визначається довжина повідомлення, на якій досягається така пропускна здатність. Константа `nMax`, яка передається до програми з командного рядка, задає обмеження на максимальну довжину повідомлення, що пересилається. А константа `nTimes`, яка теж задається користувачем в командному рядку, визначає кількість повторень для усереднення результату. Відсутність цих параметрів, або їхній не числовий формат викликає вкидання винятків.

У програмі для визначення латентності системи спочатку пересилається повідомлення нульової довжини і виміряється час потрібний для цього. Потім, починаючи з пересилання одного елемента типа `double`, довжина повідомлень поступово подвоюється, поки не перевищить значення `nMax`. Одночасно провадиться вимірювання часу, який затрачується на обмін пакетами між двох процесів. Для часових вимірювань, з погляду на їхню короткочасність, застосовується метод `currentTimeMillis` класу `System`, а не метод `Wtime` класу MPI. Вимі-

ряний час з мілісекунд переводиться у секунди шляхом множення отриманого значення на 1000.

```
import mpi.*;
class Ex_08 {
  private int nMax, nTimes, sizeOfDouble;
  private final int DATA_HEAD = 8; // Довжина заголовка пакета
  double a[];
  private static int rank;
  private static String[] arg;
  public Ex_08(String[] args) {
    arg = MPI.Init(args);
    rank = MPI.COMM_WORLD.Rank();
    nMax = Integer.parseInt(arg[0]);
    nTimes = Integer.parseInt(arg[1]);
    a = new double[nMax];
    sizeOfDouble = MPI.COMM_WORLD.Pack_size(1, MPI.DOUBLE) -
                   DATA_HEAD;
  }
  private void run() {
    int messageSize = 0, lmax = 0, n = 0;
    double max = 0.0, timeStart, time, bandwidth;
    do {
      timeStart = System.currentTimeMillis();
      for (int i = 1; i <= nTimes; i++) {
        if (rank == 0) {
          MPI.COMM_WORLD.Send(a, 0, n, MPI.DOUBLE, 1, 1);
          MPI.COMM_WORLD.Recv(a, 0, n, MPI.DOUBLE, 1, 1); }
        if (rank == 1) {
          MPI.COMM_WORLD.Recv(a, 0, n, MPI.DOUBLE, 0, 1);
          MPI.COMM_WORLD.Send(a, 0, n, MPI.DOUBLE, 0, 1); }
      }
      time = (System.currentTimeMillis() - timeStart) /
             2 / nTimes;
      bandwidth = ((double)messageSize / Math.pow(2, 20)) /
                  time * 1000;
      if (max <= bandwidth) {
        max = bandwidth;
        lmax = messageSize; }
      if (rank == 0) {
        if(n == 0)
          System.out.printf(" %nЛатентність системи = %f" +
```

```
                              " секунд%n %n", time / 1000);
          else
            System.out.printf("При розмірі повідомлення %6d" +
                              " байтів пропускна здатність =" +
                              "%7.2f МіБ/с%n", messageSize,
                              bandwidth);
        }
        n = (n == 0) ? 1 : 2 * n;
        messageSize = sizeOfDouble * n; }
      while(messageSize <= nMax);
      if(rank == 0)
        System.out.printf(" %nМаксимальна пропускна здатність" +
                          " =%7.2f МіБ/с, при розмірі" +
                          " повідомлення =%6d байтів", max, lmax);
    }
    public static void main(String args[]) throws MPIException {
      try {
        Ex_08 ex = new Ex_08(args);
        ex.run();
      }
      catch (ArrayIndexOutOfBoundsException e) {
        if (rank == 0) System.err.printf(
          " %nНе достатньо параметрів для методу main." +
          "%nСинтаксис команди запуску програми:%n" +
          " %n\t\tmpjrun.sh -np <К-ВО_ПРОЦ.> Ex_8 " +
          "<МАКС_РОЗМІР_ПАКЕТУ> <К-ТЬ_ЦИКЛІВ>");
      }
      catch (NumberFormatException e) {
        if (rank == 0) System.err.printf(
          " %nОбидва параметри програми повинні бути числами.%n" +
          "Наприклад, такі: 1024 і 100000, а не \"%s\" і \"%s\".",
          arg[0], arg[1]);
      }
      catch (MPIException e) {
        if (rank == 0) System.err.printf(
          " %nПомилковий параметр \"-np %s\" у командному" +
          " рядку.%nКількість процесів повинна бути не менша" +
          " за 2.", args[1]);
      }
      finally { MPI.Finalize(); }
      if (rank == 0) System.out.printf("%n %n");
    }
  }
```

5.3.2 Передача/прийом повідомлень без блокування

Повідомлення з блокуванням за рахунок очікування деякої умови для завершення операції залишаються на деякий час незавершеними, це призводить до сповільнення роботи всієї програми. Для усунення цього явища в MPI передбачений набір методів для здійснення ***асинхронної передачі*** даних. На відміну від методів, які спричиняють блокування процесів, повернення з методів даної групи відбувається відразу після їхнього виклику без будь-якої зупинки роботи процесів, що до них звернулися. На тлі подальшого виконання програми одночасно відбувається й обробка асинхронно запущеної операції. Насправді, при створенні програми програміст завжди знає, що в деякий момент йому буде потрібен масив, який у цей час обчислюється іншим процесом. Тому розроблювач програми може заздалегідь виставити в програмі асинхронний запит на одержання даного масиву, а до того моменту, коли масив реально буде потрібний, або підготовлений для подальшого обчислення, програма може виконувати будь-яку іншу корисну роботу. Власно кажучи, у багатьох випадках зовсім не обов'язково чекати закінчення пересилання повідомлення для виконання наступних обчислень. У принципі, дана можливість є винятково корисною для створення ефективних програм. Однак, для дійсного завершення асинхронного обміну ще потребується виклик додаткового методу, *призначенням якого є або перевіряння факту завершення операції*, або безпосереднє очікування її завершення. Тільки після цього можна використовувати буфер пересилання для інших цілей без побоювання зіпсувати повідомлення, що було переслане до іншого процесу.

Може здатись, що якщо є можливість ховати операції прийому/передачі повідомлень на тлі поточних обчислень, то цим треба беззастережно користуватися. Однак на практиці не все добре узгоджується з теорією. Багато чого залежить від конкретної реалізації тієї чи іншої бібліотеки MPI. До того ж, на жаль, далеко не завжди асинхронні операції ефективно підтримуються на апаратному рівні комп'ютера та й системним оточенням, яке встановлено на ньому. Тому не

слід дивуватися, якщо ефект від виконання обчислень на тлі пересилань виявиться нульовим або зовсім невеликим. Проте зроблені зауваження стосуються тільки питань ефективності. А от стосовно функціональності, яка досягається при програмуванні за допомогою асинхронних операції, особливо слід підкреслити їхню виняткову корисність. Це призводить до того, що таки операції присутні практично в кожній реальній програмі.

Як і при розгляді методів прийому/передачі повідомлень з блокуванням процесів, є сенс розгляд операцій роботи з повідомленнями без їхнього блокування розпочати з методів пересилання. Метод класу `Comm` – аналог процедури `MPI_ISEND`, який запускає стандартний обмін даними без блокування це –

```
public Request Isend(Object buf, int offset, int count,
                     Datatype datatype, int dest, int tag)
                                        throws MPIException
```

Типи і кількість параметрів, які передаються у метод, ні чим не відрізняється від типів і кількості параметрів, що й у методі пересилання даних з блокуванням `Send`, і призначення параметрів таке саме. Але на відміну від операції з блокуванням, яка не повертає ніякого значення, цей метод вже повертає посилання на об'єкт запиту типа `Request`. Повертання такого значення пов'язане з тим, що *повернення з методу відбувається відразу після ініціалізації процесу передачі* без очікування обробки всього повідомлення, яке перебуває в буфері. Така поведінка означає, що в ніякому разі не можна повторно використовувати даний буфер для інших цілей без одержання додаткової інформації, яка підтверджує дійсне завершення поточного повідомлення. Для визначення того моменту часу, коли можна повторно використовувати буфер `buf` без побоювання зіпсувати повідомлення, що передається, служить низка методів класу `Request`, які опрацьовують об'єкти, які й повертаються даним методом.

За аналогією з трьома модифікаціями методу `Send`, в класі `Comm` передбачені три додаткових варіанти методу `Isend`:

```
public Request Ibsend(Object buf, int offset, int count,
```

```
                        Datatype datatype, int dest, int tag)
                                                throws MPIException
```

− асинхронне пересилання повідомлення з буферуванням, аналог процедури MPI_IBSEND;

```
  public Request Issend (Object buf, int offset, int count,
                        Datatype datatype, int dest, int tag)
                                                throws MPIException
```

− асинхронне пересилання із синхронізацією, аналог процедури MPI_ISSEND;

```
  public Request Irsend(Object buf, int offset, int count,
                        Datatype datatype, int dest, int tag)
                                                throws MPIException
```

− асинхронне пересилання повідомлення за готовністю, аналог процедури MPI_IRSEND.

Семантика роботи всіх цих трьох методів, як було вказано вище, передбачає відсутність блокування процесів під час виконання операцій.

Для одержання повідомлення, як і в попередньому випадку операцій з блокуванням, є лише один метод, який є аналогом процедури MPI_IRECV і має наступне оголошення:

```
  public Request Irecv(Object buf, int offset, int count,
                       Datatype datatype, int src, int tag)
                                                throws MPIException
```

Типи, кількість і призначення параметрів цього методу теж повністю відповідає типам, кількості і призначенню параметрів методу Recv. Але на відміну від останнього, завдяки тому, що цей метод здійснює асинхронний прийом без блокування процесу і повернення з методу відбувається відразу після ініціалізації прийому без очікування одержання всього повідомлення і його запису у буфер, метод вже повертає посилання на об’єкт класу Request, а не Status. За-

кінчення процесу прийому знов таки можна визначити за допомогою методів класу `Request`, надіславши відповідне повідомлення до отриманого об'єкта того ж класу.

Незалежно від того, відправлене повідомлення методом `Send` *або* `Isend` *або навіть кожним з шістьох модифікованих методів воно може бути прийнятим, як методом* `Recv`, *так і методом* `Irecv`.

Ще раз необхідно звернути особливу увагу на той факт, що до підтвердження повного завершення асинхронної операції відповідними додатковими методами класу `Request`, не в якому разі не можна записувати нові дані до масиву, який використовується як буфер!

За аналогією з методом з блокуванням `Probe`, який надає можливість одержання з об'єкту типа `Status` інформації про структуру очікуваного повідомлення з ідентифікатором `tag` від процесу з номером `source`, існує відповідний асинхронний метод (є аналогом `MPI_IPROBE`) і має наступне оголошення:

```
public Status Iprobe(int source, int tag) throws MPIException
```

Якщо повідомлення, інформація про яке запрошується методом `Iprobe` вже є доступним, то метод так само повертає об'єкт класу `Status`, як це мало місце при застосуванні методів `Recv` або `Probe`. В іншому випадку, якщо повідомлення із зазначеними атрибутами ще не дійшло до одержувача, повертається значення `null`.

Безпосереднє очікування завершення всякої асинхронної операції, яка асоційована з ідентифікатором об'єкта типа `Request`, що повертається після виклику методу `Isend`, трьох його модифікацій або методу `Irecv`, виконується за допомогою методу класу `Request`, який є аналогічним до процедури `MPI_WAIT` і має таке оголошення:

```
public Status Wait() throws MPIException
```

Поки асинхронна операція, що асоційована з об'єктом, якому надсилається повідомлення `Wait`, *не буде завершена*, *процес*, *який викликав метод*, *буде знаходитись у заблокованому стані*. Повернення об'єкта типа `Status`, з якого можна дізнатися про параметри повідомлення, має сенс тільки в тому випадку, коли з об'єктом типа `Request` асоційована асинхронна операція прийому. Після виконання метода `Wait`, асоційованого з об'єктом типа `Request`, останньому привласнюється значення `REQUEST_NULL`, яке є статичним полем класу MPI.

Метод `Wait` дозволяє очікувати завершення лише одній операції прийому/передачі, яка пов'язана з деяким конкретним об'єктом типа `Request`. Однак часто виникає потреба в обробці завершення декількох таких операцій. Тому у класі `Request` передбачені ще декілька статичних методів, які розширюють функціональність методу `Wait`. Це такі методи, як: `Waitall, Waitany i Waitsome` – аналоги стандартних процедур `MPI_WAITALL`, `MPI_WAITANY` та `MPI_WAITSOME`, відповідно.

Метод з оголошенням:

```
public static Status[] Waitall(Request[] req)
                                        throws MPIException
```

здійснює блокування процесу, що його викликав, і *очікує завершення всіх асинхронних операцій*, асоційованих з активними запитами в масиві `req` об'єктів типа `Request`. Размер массива об'єктів типа `Status`, який повертається методом, *повинен бути того ж самого розміру*, що й масив запитів `req`. При поверненні з методу запити, які були оброблені стають неактивними і одержують значення `REQUEST_NULL`. Якщо при виклику методу масив `r` містив деякі неактивні запити, то відповідні елементи результуючого масиву одержують посилання зі значенням `null`.

У прикладі програми, що наведений нижче (Ex_09), всі процеси обмінюються повідомленнями, уміст яких відповідає рангу процесу відправнику, з найближчими сусідами відповідно до топології кільця за допомогою асинхронних

операцій. Необхідно відзначити, що використання для цих цілей операцій з блокуванням, може призвести до виникнення так званої тупикової ситуації.

```
import mpi.*;
class Ex_09 {
  private int size, prev, next, sendBuf[] = {0},
              recvBuf[] = {0, 0};
  private static int rank;
  public Ex_09(String[] args) {
    MPI.Init(args);
    rank = MPI.COMM_WORLD.Rank();
    sendBuf[0] = rank;
    size = MPI.COMM_WORLD.Size();
    prev = (rank == 0) ? size - 1 : rank - 1;
    next = (rank == size - 1) ? 0 : rank + 1;
  }
  private void run() {
    Request[] reqs = new Request[4];
    reqs[0] = MPI.COMM_WORLD.Irecv(recvBuf, 0, 1, MPI.INT,
                                   prev, 5);
    reqs[1] = MPI.COMM_WORLD.Irecv(recvBuf, 1, 1, MPI.INT,
                                   next, 6);
    reqs[2] = MPI.COMM_WORLD.Isend(sendBuf, 0, 1, MPI.INT,
                                   prev, 6);
    reqs[3] = MPI.COMM_WORLD.Isend(sendBuf, 0, 1, MPI.INT,
                                   next, 5);
    Request.Waitall(reqs);
    System.out.printf(" %nДля процесу %d:%n  попередній " +
                      "процес = %d, а наступний = %d",
                      rank, recvBuf[0], recvBuf[1]);
  }
  public static void main(String args[]) throws MPIException {
    try {
      Ex_09 ex = new Ex_09(args);
      ex.run(); }
    catch (MPIException e) {
      ; }
    finally { MPI.Finalize(); }
    if (rank == 0) System.out.printf("%n %n");
  }
}
```

Наступне розширення методу `Wait` – це метод з таким оголошенням:

```
public static Status Waitany(Request[] req) throws MPIException
```

На відміну від попереднього методу, цей метод здійснює *очікування завершення хоча б однієї з активних асинхронних операцій*, асоційованих з елементами масиву запитів `req`. Якщо на момент виклику завершилися декілька з очікуваних операцій, то випадковим чином обирається лише одна з них. Яка саме операція обрана можна дізнатися з об'єкта, який повертається методом. В полях `source` і `tag` цього об'єкта, як і завжди, містяться дані про ранг процесу і тег повідомлення, а у полі `index` повертається номер запиту у масиві `req`, який відповідає операції, що завершилась. Після виконання методу відповідний елемент масиву `req` стає неактивним і йому присвоюється значення `REQUEST_NULL`.

Нарешті останній, третій метод з цього сімейства оголошений як:

```
public static Status[] Waitsome(Request[] req)
                                        throws MPIException
```

очікує завершення декількох або хоча б однієї з активних асинхронних операцій, асоційованих з ідентифікаторами масиву `r`. *Розмір масиву об'єктів* `Status`, що повертається методом, *дорівнює кількості завершених асинхронних операцій*. Індекси в масиві запитів для операцій, які завершились, можна дізнатися з полів `index` об'єктів масиву типа `Status`, який повертається методом. Відповідні єлементи масиву запитів `r` стають неактивними і одержують значення `REQUEST_NULL`.

Наступній приклад (Ex_10) демонструє можливості використання методу `Waitsome` для організації комунікаційної схеми «`master-slave`» – це доволі розповсюджена організація обчислювальних процесів, коли всі процеси спілкуються з одним виділеним процесом. У прикладі всі процеси, крім процесу 0, на кожній ітерації циклу обчислюють за допомогою виклику методу `slave` свою локальну частину масиву `a`, після чого посилають її головному процесу. Процес

0 спочатку ініціює прийом, без блокування, повідомлень від всіх інших процесів, після чого чекає приходу хоча б одного з них. Для повідомлень, що надійшли, процес 0 викликає метод обробки `master`, після чого знов ініціює отримання, повідомлень без блокування. Таким чином, процес 0 обробляє тільки ті порції даних, які готові на даний момент. При цьому для коректності роботи програми потрібно забезпечити, щоб процес 0 устигав обробити повідомлення, котрі надходять до нього, тобто, щоб метод `slave` працював значно довше методу `master` (у противному випадку створення паралельної програми не має особливого сенсу). Крім того, у прикладі використовується нескінченний цикл, тому у подібній конкретній програмі потрібно передбачити дійсні умови завершення роботи програми.

```
import mpi.*;
class Ex_10 {
  public static void main(String args[]) throws MPIException {
    Ex_10 ex = new Ex_10();
    int rank, size;
    String[] st = MPI.Init(args);
    int N = Integer.parseInt(st[0]);
    rank = MPI.COMM_WORLD.Rank();
    size = MPI.COMM_WORLD.Size();
    int root = size - 1, rows = size - 1;
    Request[] reqs = new Request[rows];
    Status statuses[];
    double[][] a = new double[rows][N];
    if(rank != root)
      while(true) {
        ex.slave(a[rank], N);
        MPI.COMM_WORLD.Send(a[rank], 0, N, MPI.DOUBLE, root, 5);
      }
    else {
      for(int i = 0; i < rows; i++)
        reqs[i] = MPI.COMM_WORLD.Irecv(a[i], 0, N, MPI.DOUBLE,
                                       MPI.ANY_SOURCE, 5);
      while(true) {
        statuses = Request.Waitsome(reqs);
        for(int i = 0; i < statuses.length; i++) {
          ex.master(a[statuses[i].index], N);
```

```
            reqs[statuses[i].index] = MPI.COMM_WORLD.
                                  Irecv(a[statuses[i].index], 0, N,
                                  MPI.DOUBLE, indexes[i], 5);
          }
        }
      }
      MPI.Finalize();
    }

  void slave(double[] a, int n) {
  // Обробка локальної частини масива a
    }

    void master(double[] a, int n) {
  // Обробка одержаної частини масива a
    }
  }
```

Всі методи сімейства `Wait` блокують процес, який них викликав, доти, поки не буде виконана неявна умова їхнього завершення. Але така їхня поведінка сповільнює виконання програми, тому в класі `Request` передбачено ще одне сімейство подібних методів, які функціонують так само, як і методи сімейства `Wait`, але не блокують процеси. Це чотири методи-аналоги стандартних процедур `MPI_TEST`, `MPI_TESTALL`, `MPI_TESTANY` і `MPI_TESTSOME` з наступними оголошеннями:

```
public Status Test() throws MPIException
```

виконує ті ж самі дії, що й `Wait`, однак вихід з нього виконується одразу після виклику;

```
public static Status[] Testall (Request[] req)
                                        throws MPIException
```

працює так само, як і `Waitall`, але не блокує процес;

```
public static Status Testany (Request[] req)
```

```
                                        throws MPIException
```

це асинхронний метод, який відповідає методу Waitany і не блокує процес;

```
public static Status[] Testsome (Request[] req)
                                        throws MPIException
```

повний аналог методу Waitsome без блокіровки процесів і сповільнення програми.

Наступний приклад (Ex_11) демонструє застосування операцій без блокування для реалізації транспонування квадратної матриці, розподіленої між процесами по рядках. Спочатку кожен процес локально розраховує nl рядків матриці a (для цього використовується метод fillMatrix) Після завершення розрахунків вихідна матриця a роздруковується на екрані термінала (метод printMatrix). Потім за допомогою асинхронних операцій Isend та Irecv ініціюються всі необхідні для транспонування обміни даними. На тлі обмінів, що починаються, кожен процес транспонує свою локальну частину масиву a. Після цього процеси за допомогою виклику методу Waitall чекають завершення всіх розпочатих раніш операцій асинхронного обміну даними. По завершенню отримання всіх повідомлень, кожен з процесів виконує транспонування тих, поки що нетранспонованих частин вихідної матриці, які він одержав від інших процесів. Наприкінці роздруковується отримана транспонована матриця b.

```
import mpi.*;
import java.util.*;
class Ex_11 {
  private int matrSize, size, nRows;
  private double a[][], b[][];
  private static int rank;
  private static String[] arg;
  private static Ex_11 ex;
  public Ex_11(String[] args) {
    arg = MPI.Init(args);
    rank = MPI.COMM_WORLD.Rank();
    size = MPI.COMM_WORLD.Size();
```

```
  matrSize = Integer.parseInt(arg[0]);
  if (matrSize % size != 0)
    throw new IllegalArgumentException("" + size);
  a = new double[matrSize][matrSize];
  b = new double[matrSize][matrSize];
  nRows = (matrSize - 1) / size + 1;
}
public static void main(String[] args) throws MPIException {
  boolean canDo = true;
  try { ex = new Ex_11(args); }
  catch(ArrayIndexOutOfBoundsException e) {
    canDo = false;
    if (rank == 0)
      System.err.printf(" %nНе вистачає параметра для" +
                        " методу main.%nСинтаксис команди" +
                        " запуску програми:%n %n\t\t" +
                        "mpjrun.sh -np <К-ТЬ_ПРОЦ> Ex_11" +
                        " <РОЗМІР_МАТРИЦІ>");
  }
  catch(NumberFormatException e) {
    canDo = false;
    if (rank == 0)
      System.err.printf(" %nПомилка у командному рядку.%n" +
                        "Параметр програми \"%s\" не є" +
                        " числом!", arg[0]);
  }
  catch(IllegalArgumentException e) {
    canDo = false;
    if (rank == 0)
      System.err.printf(" %nРозмірність матриці \"%s\"" +
                        " повинна бути більшою за%n" +
                        "кількість процесів \"%s\" у ціле" +
                        " число разів! %nНаприклад," +
                        " -np 4 Ex_11 8, або -np 5 Ex_11 15",
                         arg[0], e.getMessage());
  }
  finally {
    if(canDo) {
      ex.fillMatrix();
      ex.printMatrix("Вихідна матриця", ex.a);
      ex.transpose();
      ex.printMatrix("Матриця після транспонування", ex.b);
    }
```

```
      MPI.Finalize();
    }
  }
  private void transpose() {
    int i, j, k, row, col;
    double c;
    Status status;
    Request reqs[][] = new Request[nRows][size * 2];
    for(i = 0; i < size; i++)
      if(i != rank) {
        row = rank * (nRows);
        col = i * (nRows);
        for(j = 0; j < nRows; j++) {
          reqs[j][i] = MPI.COMM_WORLD.Irecv(b[row + j], col,
                              nRows, MPI.DOUBLE, i, MPI.ANY_TAG);
          reqs[j][i + size] = MPI.COMM_WORLD.Isend(a[row + j],
                                  col, nRows, MPI.DOUBLE, i, 1);}
        }
// Цикл транспонування власної частини матриці
    for(i = 0; i < nRows; i++) {
      row = i + rank * nRows;
      b[row][row] = a[row][row];
      for(j = i + 1; j < nRows; j++) {
        col = j + rank * nRows;
        b[row][col] = a[col][row];
        b[col][row] = a[row][col]; }
    }
    for(i = 0; i < nRows; i++) Request.Waitall(reqs[i]);
    row = rank * nRows;
// Транспонування частин, одержаних від інших процесів
    for(i = 0; i < size; i++) {
      if(i != rank) {
        col = i * nRows;
        for(j = 0; j < nRows; j++) {
          for(k = j + 1; k < nRows; k++) {
            c = b[row + j][col + k];
            b[row + j][col + k] = b[row + k][col + j];
            b[row + k][col + j] = c; }
        }
      }
    }
  }
  private void fillMatrix() {
```

```
      for (int i = 0; i < nRows; i++)
        for (int j = 0; j < matrSize; j++) {
          int k = i + rank * nRows;
          a[k][j] = (10 * k + j) / 10.0; }
    }
    private void printMatrix(String str, double a[][]){
      myGather(a);
      if(rank == 0) {
        System.out.printf(" %n%s%n", str);
        for(int i = 0; i < matrSize; i++)
          System.out.printf("%s%n", Arrays.toString(a[i]));
        System.out.printf(" %n"); }
    }
    private void myGather(double a[][]) {
      int i, j;
      if(rank == 0)
        for(i = 1; i < size; i++)
          for(j = 0; j < nRows; j++)
            MPI.COMM_WORLD.Recv(a[i * nRows + j], 0, matrSize,
                                MPI.DOUBLE, i, 99);
      else
        for(j = 0; j < nRows; j++)
          MPI.COMM_WORLD.Send(a[rank * nRows + j], 0, matrSize,
                              MPI.DOUBLE, 0, 99);
    }
  }
```

5.3.3 Відкладені запити на взаємодію

Методи даної групи дозволяють знизити накладні витрати, що виникають у рамках одного процесора при обробці прийому/передачі й переміщенні необхідної інформації між процесом і мережним контролером. Їхня присутність у бібліотеці MPI пов’язана з тим, що часто в програмах доводиться багаторазово виконувати обміни з однаковими параметрами, наприклад, у циклі. У цьому випадку можна один раз ініціювати операцію обміну, а потім багаторазово її запускати, не витрачаючи на кожній ітерації додаткового часу на ініціювання й підготовку відповідних внутрішніх структур даних. Крім того, у такий спосіб кілька запитів на прийом і/або передачу можуть поєднуватися разом для того, щоб

далі них можна було б запустити однією командою (втім, це зовсім необов'язково є добрим, оскільки може призвести до перевантаження комунікаційної мережі).

При застосуванні відкладених запитів на взаємодію, *спосіб прийому повідомлення ніяк не залежить від способу його пересилання*: повідомлення, відправлене за допомогою відкладених запитів або звичайним способом, може бути прийняте, як звичайним способом, так і за допомогою відкладених запитів.

Відкладений запит на відправлення повідомлення – аналог стандартної процедури `MPI_SEND_INIT` оголошується у класі `Comm` і це оголошення має наступний вигляд:

```
public Prequest Send_init(Object buf, int offset, int count,
                          Datatype datatype, int dest, int tag)
                                                 throws MPIException
```

Як видно з оголошення цей метод приймає ті ж самі параметри і з тією ж метою, що і метод `Send`. Але, на відміну від останнього, *повернення з методу виконується одразу після звернення до нього*, *а процес відправлення повідомлення не починаючись відкладається до певного моменту у майбутньому*. Крім того метод повертає посилання на об'єкт відкладеного запиту типа `Prequest`, а не класу `Status`. Однак ще раз необхідно наголосити на тому, що *основна відміна методу полягає в затриманні початку відправлення повідомлення* одержувачу!

За аналогією до трьох модифікацій кожного з методів `Send` і `Isend`, в класі `Comm` передбачені й три додаткових варіанти методу `Send_init`:

```
public Prequest Bsend_init(Object buf, int offset, int count,
                           Datatype datatype, int dest, int tag)
                                                 throws MPIException
```

формування відкладеного запиту на передачу повідомлення з буферуванням;

```
public Prequest Ssend_init(Object buf, int offset, int count,
                           Datatype datatype, int dest, int tag)
```

```
                                                    throws MPIException
```

формування відкладеного запиту на передачу повідомлення із синхронізацією;

```
public Prequest Rsend_init(Object buf, int offset, int count,
                           Datatype datatype, int dest, int tag)
                                                    throws MPIException
```

формування відкладеного запиту на передачу повідомлення за готовністю.

Знов таки дія цих методів відповідає дії стандартних аналогів бібліотек MPI – `MPI_BSEND_INIT`, `MPI_SSEND_INIT` та `MPI_RSEND_INIT`.

Відкладений запит на прийом повідомлення здійснюється за допомогою наступного методу класе `Comm`, який є аналогом процедури `MPI_RECV_INIT`:

```
public Prequest Recv_init(Object buf, int offset, int count,
                          Datatype datatype, int source, int tag)
                                                    throws MPIException
```

При виклику цього методу відбувається формування відкладеного запиту на прийом повідомлення. І знов, як і у випадку з методом `Send_init`, повернення з методу виконується негайно, але сама *операція прийому повідомлення не починається*!

Для керування моментом ініціалізації і початку виконання операцій прийому/відправлення повідомлень з відкладеними запитами, які були підготовлені раніше, у класі `Prequest` передбачені два методи-аналоги процедур `MPI_START` і `MPI_STARTALL`. Перший з них – метод з оголошенням:

```
public void Start () throws MPIException
```

запускає операцію обміну, яка відповідає об’єкту відкладеного запиту, до якого відправлено повідомлення цього методу. Комунікація завершується так саме як і виконанні методу `Wait` або `Test` при керуванні асинхронними операціями без блокування, але при завершенні операції передачі повідомлення, *відкладений*

запит на операцію не переводиться у неактивний стан і може бути знов ініційований шляхом звернення до методу `Start`.

Другий метод, який має оголошення:

```
public static void Startall(Prequest[] array_of_request)
                                        throws MPIException
```

ініціює низку відкладених запитів на виконання операцій обміну, що відповідають значенням елементів масиву тиру `Prequest`, який передається до метода як параметр. Всі операції теж запускаються як асинхронні операції без блокування. Як і в попередньому випадку метод `Startall` може багаторазово ініціювати операції комунікацій з відкладеними запитами, використовуючи той самий масив запитів

На відміну від звичайних бібліотек MPI, *в класі* `Prequest` *відсутній аналог процедури* `FREE`, за допомогою якої програміст повинен видаляти відкладені запити на взаємодію після їхнього використання. В мові Java така операція не має сенсу, оскільки видалення об'єктів, посилання на яки вже не потрібні, виконує збирач сміття.

У наступному фрагменті програми ініціюються відкладені запити на операції обидваспрямованого обміну із сусідніми процесами у кільцевій топології. Самі операції багаторазово запускаються на кожній наступній ітерації циклу. По завершенні циклу відкладені запити видаляти не потрібно, хоча, при необхідності, можна скористуватися методом `finalize`, який успадкований від батьківського класу `Request`.

```
prev = rank == 0 ? size - 1 : rank - 1;
next = rank == size - 1 ? 0 : rank + 1;
preqs[1] = MPI.COMM_WORLD.Recv_init(rbuf, 0, 1, MPI.DOUBLE, prev, 5);
preqs[2] = MPI.COMM_WORLD.Recv_init(rbuf, 1, 1, MPI.DOUBLE, next, 6);
preqs[3] = MPI.COMM_WORLD.Send_init(sbuf, 0, 1, MPI.DOUBLE, prev, 6);
preqs[4] = MPI.COMM_WORLD.Send_init(sbuf, 1, 1, MPI.DOUBLE, next, 5);
for(i = ...
  {
    sbuf[1] = ...
```

```
    sbuf[2] = ...
    Prequest.Startall(preqs);
// ...
    stats = Prequest.Waitall(preqs);
// ...
  }
```

5.3.4 Виникнення тупикових ситуацій (deadlock) і способи їхнього усунення

Використання методів прийому й пересилання з блокуванням пов'язане з можливим виникненням, так званих, ***тупикових ситуацій***. Виникнення таких ситуацій може спостерігатись, наприклад, при роботі двох паралельних процесів, яки повинні обмінятися даними.

Якщо у програмі зустрінеться наведений нижче уривок програми, то у програмі неодмінно станеться тупикова ситуація. Тут два процеси обмінюються один з одним числами. Однак через те, що обидва процеси спочатку використовують функції прийому повідомлення з блокуванням, кожен з процесів зависає чекаючи повідомлення, яке так ніколи й не надійде.

```
int[] j = {0}, i = {(rank + 1)*5};
if (rank == 0)
  {
    status = MPI.COMM_WORLD.Recv(j, 0, 1, MPI.INT, 1, message2);
    MPI.COMM_WORLD.Send(i, 0, 1, MPI.INT, 1, message1);
  }
if (rank == 1)
  {
    status = MPI.COMM_WORLD.Recv(j, 1, MPI.INT, 0, message1);
    MPI.COMM_WORLD.Send(i, 0, 1, MPI.INT, 0, message2);
  }
```

Схематично таку найгіршу ситуацію, коли обидва процеси спочатку виконують синхронний метод прийому повідомлення `Recv` і лише потім звертаються до методу посилання даних можна зобразити наступною схемою:

процес 0:	процес 1:
`Recv` від процесу 1 `Send` до процесу 1	`Recv` від процесу 0 `Send` до процесу 0

Завжди виникає тупикова ситуація!

В цьому випадку тупик виникає навіть незалежно від способу реалізації методів прийому/передачі даних.

Здається, що було б цілком природно в кожному процесі спочатку скористатися методом `Send`, а потім методом `Recv`. Умовно така ситуація показана нижче.

процес 0:	процес 1:
`Send` до процесу 1 `Recv` від процесу 1	`Send` до процесу 0 `Recv` від процесу 0

Може виникнути тупикова ситуація!

Але саме цього теж не варто робити. Ситуація, коли обидва процеси спочатку посилають один одному повідомлення, а потім приймають, може спричинити, а може й не призвести до тупикової ситуації. У випадку, коли обмін виконується з буферуванням, тобто процес блокується лише доти, поки повідомлення копіюється в системний буфер, тупикової ситуації не виникає. Якщо ж відправлення й прийом у даній реалізації MPI здійснюються синхронно, відбудеться зависання. Справа в тому, що заздалегідь не відомо, як в тій або іншої бібліотеці MPI реалізований метод `Send`. Якщо розроблювачі для гарантії коректного повторного використання буферу пересилання заклали схему, при якій процес, що посилає повідомлення, чекає початку прийому, то виникає класична тупикова ситуація. Перший процес не може повернутися із методу пересилання, оскільки другий не починає прийом повідомлення. А другий процес не може почати прийом повідомлення, оскільки сам зі схожої причини застряг на посилці.

У зв’язку з можливістю виникнення таких аварійних ситуацій, коли програма нібито працює, але насправді процеси знаходяться в стані очікування реакції один-одного, є сенс розглянути різні засоби запобігання тупикових ситуацій.

По-перше найпростішим варіантом запобігання тупикової ситуації є замі-

на порядку проходження методів пересилання/прийому повідомлення на одному із процесів, наприклад, так, як це зображено на схемі спілкування двох процесів, що наведена нижче.

процес 0:	процес 1:
`Send` до процесу 1 `Recv` від процесу 1	`Recv` від процесу 0 `Send` до процесу 0

Тупикова ситуація ніколи не виникає!

Тобто гарантовано уникнути тупикової ситуації допомагає наступна зміна в програмі:

```
int[] j = {0}, i = {(rank + 1)*5};
if (rank == 0)
  {
    MPI.COMM_WORLD.Send(i, 0, 1, MPI.INT, 1, message1);
    status = MPI.COMM_WORLD.Recv(j, 0, 1, MPI.INT, 1, message2);
  }
if (rank == 1)
  {
    status = MPI.COMM_WORLD.Recv(j, 1, MPI.INT, 0, message1);
    MPI.COMM_WORLD.Send(i, 0, 1, MPI.INT, 0, message2);
  }
```

Іншим варіантом запобігання виникненню тупикової ситуації може стати використання операцій без блокування для здійснення комунікаційних операцій. Для цього треба лише замінити виклик методу прийому повідомлення з блокуванням на звернення до методу асинхронного прийому `Irecv` і розташувати його перед викликом методу `Send`. Тобто схема комунікації буде зараз виглядати в такий спосіб:

процес 0:	процес 1:
`Send` до процесу 1 `Recv` від процесу 1	`Irecv` від процесу 0 `Send` до процесу 0 `Wait`

Тупикова ситуація ніколи не виникає!

У такій послідовності дій тупикова ситуація гарантовано не виникає,

оскільки до моменту виклику методу Send у першому процесі запит на прийом повідомлення вже буде виставлений, а тому, нульовий процес може починати передачу даних. При цьому *виклик методу* Irecv у програмі *рекомендується виконувати якомога раніше*, щоб тім самим заздалегідь надати можливість начати пересилання й максимально використати переваги асинхронності.

Спеціально для усунення тупикових ситуацій у клас Comm пакета добавлений окремий, незв'язаний з іншими методами класу, метод-аналог процедури MPI_SENDRECV, з таким оголошенням:

```
public Status Sendrecv(Object sendbuf, int sendoffset, int sendcount,
                       Datatype sendtype, int dest, int sendtag,
                       Object recvbuf, int recvoffset, int recvcount,
                       Datatype recvtype, int source, int recvtag)
                                                  throws MPIException
```

За допомогою цього методу процес, що його викликав, виконує одночасний прийом/передачу повідомлень з блокуванням. При виклику даного методу здійснюється посилка sendcount елементів типа sendtype з масиву sendbuf починаючи з позиції sendoffset процесу з номером dest і з тегом sendtag у активному комунікаторі. Одночасно з цим виконується прийом іншого повідомлення довжиною recvcount елементів, починаючи з позиції recvoffset, до буферу recvbuf. Тип елементів, які приймаються, визначається параметром recvtype, а відправник і тег повідомлення значенням параметрів source і recvtag, відповідно. Для ідентифікації прийнятого повідомлення повертається об'єкт типа Status. Даний метод *дозволяє використовувати у якості процесів відправника і приймача один й той самий процес, але буфери передачі й прийому даних не повинні перетинатися, тобто повинні бути різними*. Стандарт MPI передбачає, що *будь які модифікації бібліотек повинні гарантувати відсутність виникнення тупикових ситуацій при використанні цього методу*. Крім того оговорюється, що повідомлення, яке відправлене методом Sendrecv, може бути прийнято звичайним методом Recv і, навпаки, методом Sendrecv можна обробити повідомлення, яке відправлене методом Send.

Іноді буває незручно використовувати різні буфери прийому/пересилання повідомлення, як цього вимагає метод `Sendrecv`. Тому в класі `Comm` пакета передбачений подібний метод (аналог стандартного методу `MPI_SENDRECV_REPLACE`), якій дозволяє здійснювати прийом/передачу повідомлення у той самий буфер. Оголошення методу має вигляд:

```
public Status Sendrecv_replace(Object buf, int offset, int count,
                               Datatype datatype, int dest,
                               int sendtag, int source,
                               int recvtag)
                                          throws MPIException
```

Метод виконує прийом/передачу повідомлень із блокуванням через загальний буфер `buf`. Цілком зрозуміло, що в даному разі на відміну від попереднього, *прийняте повідомлення не повинно перевищувати за розміром те, що відправляється*, а передані й прийняті дані повинні бути одного типа. Метод також гарантує відсутність тупикової ситуації.

Наведений нижче приклад (Ex_12), як і описаний раніш (див. стор. 108), демонструє виконання операції обидваспрямованого обміну із сусідніми процесами в кільцевій топології. Кожен процес відправляє свій ранг до сусідів праворуч – `next` і ліворуч – `prev`. Однак робиться це за допомогою двох викликів методів `Sendrecv`. При цьому гарантовано не виникає тупикової ситуації.

```
import mpi.*;
class Ex_12 {
  private int rank, size, prev, next, sendBuf[] = {0},
              recvBuf[] = {0, 0};
  public Ex_12(String[] args) {
    MPI.Init(args);
    size = MPI.COMM_WORLD.Size();
    rank = MPI.COMM_WORLD.Rank();
    sendBuf[0] = rank;
    prev = rank == 0 ? size - 1 : rank - 1;
    next = rank == size - 1 ? 0 : rank + 1;
  }
  private void run() {
```

```
      MPI.COMM_WORLD.Sendrecv(sendBuf, 0, 1, MPI.INT, prev, 6,
                              recvBuf, 1, 1, MPI.INT, next, 6);
      MPI.COMM_WORLD.Sendrecv(sendBuf, 0, 1, MPI.INT, next, 5,
                              recvBuf, 0, 1, MPI.INT, prev, 5);
      System.out.printf(" %nДля процесу %d:%n  попередній " +
                        "процес = %d, а наступний = %d",
                        rank, recvBuf[0], recvBuf[1]);
    }
    public static void main(String[] args) throws MPIException {
      Ex_12 ex = new Ex_12(args);
      ex.run();
      MPI.Finalize();
      if (ex.rank == 0) System.out.printf("%n %n");
    }
  }
```

5.3.5 Контрольні запитання та завдання для самостійної роботи

− На які дві основні групи поділяються всі методи передачі повідомлень в системах MPI? Дайте стислу характеристику цих груп.

− До об’єктів яких класів належать методи, що надсилають і приймають повідомлення в системі передачі повідомлень MPJ Express?

− Які параметри системи обов’язково повинні бути вказані процесом-відправником при звертанні до методу відправки повідомлення? Які дії у відповідь повинен зробити процес-одержувач і з якими параметрами системи?

− Назвіть дві основні групи методів типа «точка-точка», охарактеризуйте них.

− Розкажіть про три етапи виконання операцій обміну типа точка-точка у стандартному режимі. Які поля об’єкту типа `Status` задіяні у цих операціях?

− Докладно розкажіть про операцію відправлення повідомлення з блокуванням. Яким є призначення параметрів, що передаються до нього?

− Що гарантується стандартом MPI при відправленні/прийомі повідомлень за допомогою операцій з блокуванням?

− Які константи типа `Datatype` оголошені у класі MPI, охарактеризуйте константи типів, які не мають аналогів серед примітивних типів мови Java.

– Як система реагує на відправлення повідомлення до процесу MPI.PROC_NULL?

– Чим дія методу Bsend відрізняється від дії методу Send? Які попередні дії повинен зробити програміст до відправлення повідомлення методом Bsend?

– Охарактеризуйте методи відправки повідомлень з блокуванням, що залишилися (Ssend та Rsend). Що означає завершення операції для різних видів пересилання даних із блокуванням?

– Які методи передбачає стандарт MPI для отримання повідомлень з блокуванням? Які особливості властиві методу Recv?

– Як визначити реальні атрибути прийнятого повідомлення, якщо воно отримувалося з атрибутами MPI.ANY_SOURCE і MPI.ANY_TAG?

– Що можна сказати відносно вимог стандарту MPI про порядок отримання повідомлень деяким процесом.

– Для яких цілей призначені методи Get_count і Probe? Як і коли них використовують?

– Які вади властиві операціям с блокуванням процесів і як асинхронні операції них усувають? Поясніть цей механізм.

– Що потребується для дійсного завершення асинхронного обміну між двома процесами?

– Назвіть асинхронні методи відправки повідомлень у мережі. Об'єкт якого класу повертають ці методи, для чого потрібні таки об'єкти?

– Як виконується асинхронний прийом повідомлень? Чи можна повідомлення, що відправлене за допомогою операції пересилання з блокуванням, прийняти асинхронною операцією прийому і навпаки?

– Як можна визначити атрибути асинхронного повідомлення, якщо метод Irecv повертає об'єкт класу Request, а не Status?

– Розкажіть про сімейство методів Wait. Скільки їх, для чого вони потрібні, чим відрізняються один від одного?

– Як можна позбавитися блокування процесів при звертанні до методів сімейства Wait? розкажіть про методи, які усувають цей недолік.

– Що таке відкладені запити на взаємодію і для чого вони потрібні?

– Розкажіть про сімейство методів відправлення і прийому повідомлень, яке забезпечує відкладені запити на взаємодію.

– Яким же чином все ж таки відбувається запуск операцій відкладених запитів на взаємодію? Розкажіть про відповідні методи.

– Що таке тупикові ситуації, який механізм їхньої появи у паралельних програмах?

– Дайте характеристику послідовностям дій коли обов'язково виникають тупикові ситуації, коли вони можуть виникнути або ні і ситуаціям коли вони ніколи не виникають.

– Опишіть реалізацію у пакеті MPJ Express спеціальних методів стандарту MPI, які призначені для запобігання виникнення тупикових ситуацій.

– Напишіть невеличку програму, яка порівнює ефективність реалізації різних методів пересилань даних з блокуванням (`Send`, `Bsend`, `Ssend`, `Rsend`) між двома виділеними процесорами.

– Спробуйте написати програму, яка порівнює ефективність реалізації пересилань даних між двома виділеними процесорами синхронними і асинхронними методами.

– Подумайте яким чином можна за допомогою метода `Test` змоделювати функціонування метода `Wait`. Зробить таке моделювання.

– З огляду на відмінності у використанні методів `Waitall`, `Waitany` і `Waitsome` спробуйте змоделювати їхнє функціонування за допомогою лише метода `Wait`.

– За допомогою пересилання повідомлень типа точка-точка спробуйте реалізувати пересилання даних між процесами по колу, за принципом передачі «естафетної палички» (черговий процес чекає повідомлення від попереднього, а потім посилає повідомлення наступному). Постарайтеся уникнути тупикової ситуації.

– Модифікуйте попередню програму так, що б пересилання даних процесами виконувалося як «зсув», тобто і посилка, і прийом повідомлень всіма процесами у системі здійснюється одночасно.

– За допомогою операцій точка-точка з блокуванням процесів напишіть програму процеси у якій спілкуються за принципом «хазяїн-підлеглий» («master-slave») – один процес розподіляє вихідні дані іншім процесам, а потім отримує від них вже перетворені дані.

– Подумайте як можна, і спробуйте реалізувати повнозв'язану топологію пересилання даних у мережі процесів, коли всі процеси відправляють до всіх інших.

– Для попередніх завдань, змінюючи обсяги даних, що пересилаються, і кількість процесів у комунікаторі дослідити ефективність таких комунікаційних схем.

– Подумайте над запитанням про можливість одержання більшої ефективності роботи програм з попередніх завдань шляхом використання відкладених запитів на взаємодію. Перевірте свої висновки на практиці, зробивши відповідні зміни у програмах.

– За допомогою методів з сімейства асинхронних операцій комунікації змоделюйте роботу метода `Sendrecv`. Порівняйте ефективність роботи Вашої моделі і власне метода.

– Спробуйте написати програму отримання скалярного добутку двох векторів, які розподілені по процесах комунікатора.

5.4 Засоби колективної комунікації процесів

За своїм смислом і організацією методи взаємодії типа точка-точка призначені для комунікації між двома виділеними процесами. Однак часто логіка паралельної програми потребує одночасної взаємодії всіх наявних в системі процесів. Створення такої програми тільки за рахунок вище описаних методів хоча й можливо, але не дуже зручно. Тому у стандарті MPI передбачена низка

методів, які дозволяють *всім процесам комунікатора одночасно брати участь в операціях* так званої ***колективної взаємодії*** процесів.

Всі колективні операції за характером взаємодії між процесами поділяються на наступні чотири групи:

1) Всі зі Всіма – всі процеси вносять свій вклад у досягнення результату операції, а потім *всі ж процеси й отримують його*. Це такі операції:
   - `Allgather`, `Allgatherv`;
   - `Alltoall`, `Alltoallv`;
   - `Allreduce`, `Reduce_scatter`;
   - `Barrier`;
2) Всі з Одним – знов-таки всі процеси вносять свій вклад у досягнення результату операції, але тільки *один кореневий процес отримує його*. У цю групу входять наступні методи:
   - `Gather`, `Gatherv`;
   - `Reduce`;
3) Один зі Всіма – один кореневий процес виконує операцію над даними, а її *результат отримують всі* процеси комунікатора. Це такі методи:
   - `Bcast`;
   - `Scatter`, `Scatterv`;
4) Інші колективні операції – операції, які не можна віднести до деякої з вище вказаних категорій:
   - `Scan`.

У пакеті MPJ Express всі ці методи є членами класу `Intracomm`, який розширює клас `Comm`, і всі вони є аналогами відповідних процедур стандартних бібліотек MPI. Методи *повинні викликатись спільно всіма процесами комунікатора*, *але набір параметрів для кожного процесу може бути різним*. Повернення з методу колективної взаємодії може відбутися в той момент, коли участь процесу в даній операції вже закінчено. Так само, як і для синхронних методів, повернення з операції означає, що вже може здійснюватись вільний доступ до буфера прийому або пересилання повідомлення.

Для колективних операцій можуть використовуватися ті ж самі комунікатори, у яких вже викликані і виконуються операцій типа точка-точка. Тобто MPI *гарантує, що повідомлення, викликані колективними операціями, ніяк не вплинуть на виконання інших операцій і не перетнуться з повідомленнями*, які з'явилися *в результаті індивідуальної взаємодії процесів*.

Власне кажучи, *при застосуванні колективних операцій стандарт* MPI ні яким чином *не обумовлює синхронізацію процесів під час взаємодії*, тому розраховувати на синхронізацію виконання процесів не можна (за винятком методу `Barrier`). Якщо якийсь процес завершив свою участь у колективній операції, то це зовсім не означає, що дана операція завершена іншими процесами комунікатора, але це не означає навіть й того, що вона ними розпочалася (якщо це можливо за змістом операції).

У колективних операціях не використовуються ідентифікатори повідомлень (теги). Таким чином, *колективні операції строго впорядковані відповідно до їхнього появлення у тексті програми*.

Розгляд колективних операцій взаємодії має сенс розпочати з методу, дія якого є найбільш простою і зрозумілою. Це метод, який є аналогом стандартної процедури `MPI_BARRIER` і має наступне оголошення:

```
public void Barrier() throws MPIException
```

З самої назви методу, є цілком зрозумілим, що він *використовується для синхронізації процесів* за допомогою механізму бар'єрів. Тобто робота всіх процесів, які приймають участь у колективному виконанні цього методу у деякому комунікаторі блокується доти, поки всі вони, не завершать виконання методу. І тільки після того, як останній процес комунікатора виконає даний метод, буде зняте блокування всіх процесів і вони продовжать виконання програми далі. Оскільки даний метод є колективним, то всі процеси повинні викликати `Barrier`, хоча реально звернення до викликів методу різними процесами комунікатора можуть бути розташовані в різних місцях програми.

У наступному прикладі (Ex_13) функціонування методу `Barrier` моделюється за допомогою відкладених запитів на взаємодію типа точка-точка і вимірюється час у мілісекундах, потрібний для цього. Для усереднення результатів робиться `nTimes` (задається як параметр програми) операцій обміну, у межах кожної з них всі процеси посилають повідомлення процесу з номером 0, після чого одержують від нього відповідний сигнал, який означає, що всі процеси дійшли до цієї точки в програмі. Використання відкладених запитів у прикладі дозволяє ініціювати посилку даних тільки один раз, а потім використовувати них на кожній ітерації циклу. Після завершення циклу, у програмі порівнюється час потрібний на моделювання процесу синхронізації з часом необхідним для синхронізації за допомогою виклику всіма процесами методу `Barrier`.

```
import mpi.*;
class Ex_13 {
  private static int rank, root = 0;
  private int size, nTimes, sendBuf[] = {0}, recvBuf[];
  private Status statuses[];
  private Prequest r0, r1, reqs[][];
  private static String[] arg;
  private static Ex_13 ex;
  public Ex_13(String[] args) {
    arg = MPI.Init(args);
    rank = MPI.COMM_WORLD.Rank();
    sendBuf[0] = rank;
    nTimes = Integer.parseInt(arg[0]);
    size = MPI.COMM_WORLD.Size();
    recvBuf = new int[size];
    reqs = new Prequest[2][size - 1];
    statuses = new Status[size - 1];
  }
  private void run() {
    double time_start, time_finish;
    if(rank == root) {
      for(int i = 0; i < size - 1; i++) {
        reqs[0][i] = MPI.COMM_WORLD.Recv_init(recvBuf, i, 0,
                                              MPI.INT, i + 1, 5);
        reqs[1][i] = MPI.COMM_WORLD.Send_init (sendBuf, 0, 0,
                                              MPI.INT, i + 1, 6);
```

```
      }
      time_start = System.currentTimeMillis();
      for(int i = 1; i <= nTimes; i++) {
        Prequest.Startall(reqs[0]);
        statuses = Prequest.Waitall(reqs[0]);
        Prequest.Startall(reqs[1]);
        statuses = Prequest.Waitall(reqs[1]); }
    }
    else {
      r0 = MPI.COMM_WORLD.Recv_init(recvBuf, 0, 0, MPI.INT,
                                    0, 6);
      r1 = MPI.COMM_WORLD.Send_init(sendBuf, 0, 0, MPI.INT,
                                    0, 5);
      time_start = System.currentTimeMillis();
      for(int i = 1; i <= nTimes; i++) {
        r1.Start();
        Status st = r1.Wait();
        r0.Start();
        st = r0.Wait(); }
    }
    time_finish = System.currentTimeMillis() - time_start;
    System.out.printf(" %nПроцес %d, час емуляції методу" +
                      " Barrier() = %f мс",
                      rank, time_finish / nTimes);
    time_start = System.currentTimeMillis();
    for(int i = 1; i <= nTimes; i++) MPI.COMM_WORLD.Barrier();
    time_finish = System.currentTimeMillis() - time_start;
    System.out.printf(" %nПроцес %d, час виконання методу" +
                      " Barrier() = %f мс",
                      rank, time_finish / nTimes);
  }
  public static void main(String[] args) throws MPIException {
    try {
      Ex_13 ex = new Ex_13(args);
      ex.run();
    }
    catch(ArrayIndexOutOfBoundsException e) {
      if (rank == root)
        System.err.printf(" %nНе вистачає параметра для" +
                          " методу main.%nСинтаксис команди" +
                          " запуску програми:%n %n\t\t" +
                          "mpjrun.sh -np <К-ТЬ_ПРОЦ> Ex_13" +
                          " <К-ТЬ_ІТЕРАЦІЙ>");
```

```
      }
      catch(NumberFormatException e) {
        if (rank == root)
          System.err.printf(" %nПомилка у командному рядку.%n" +
                            "Параметр програми \"%s\" не є" +
                            " числом!", arg[0]);
      }
      finally {
        MPI.Finalize();
        if (rank == root) System.err.printf("%n %n");
      }
    }
  }
```

Ще один доволі простий і зрозумілий колективний метод (аналог процедури `MPI_BCAST`) з оголошенням:

```
public void Bcast(Object buf, int offset, int count,
                  Datatype type, int root) throws MPIException
```

Метод *призначений для розсилання від процесу* `root` *всім процесам поточного комунікатора*, *у тому числі і самому процесу* `root`, `count` елементів даних типа `type` з масиву `buf` починаючи з позиції `offset` в ньому. При поверненні з методу вміст відповідної частини буфера `buf` процесу `root` буде скопійовано в локальні буфери кожного процесу комунікатора. *Значення параметрів* `count`, `type` *і* `root` *повинні бути однаковими у всіх процесів*, але *параметри* `buf` *і* `offset` в загальному випадку *можуть відрізнятись*.

Наприклад, для того щоб переслати від процесу з рангом 2 всім іншим процесам у комунікаторі `COMM_WORLD` застосування масив `buf`, який містить 100 цілочисельних елементів, потрібно, щоб у всіх процесах комунікатора зустрівся наступний виклик:

```
MPI.COMM_WORLD.Bcast(buf, 0, 100, MPI.INT, 2);
```

Наочно дію методу `Bcast` можна проілюструвати наступною схемою (рис. 5.1). На цій схемі, як й у подальших подібних схемах, по вертикалі зображуються різні процеси, яки беруть участь у колективній операції, а по горизонталі – блоки даних, які розташовані на них. Ці блоки можуть бути, як окремими елементами відповідних масивів даних, так і досить великими підмасивами, які містять багато елементів буферів прийому/передачі.

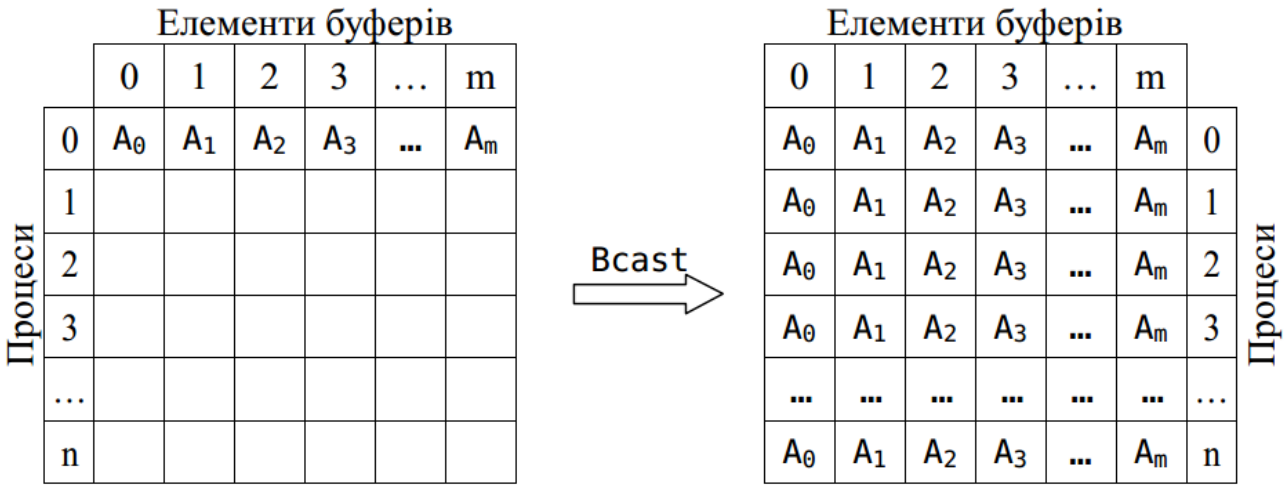


Рисунок 5.1 – Схема пересилання даних методом `Bcast`

Іншими словами при виконанні метода `Bcast` вміст буфера даних деякого розміру від одного виділеного процесу одночасно передається до буферів саме того ж розміру у всіх процесах комунікатора, такий процес пересилань ілюструється рис. 5.2

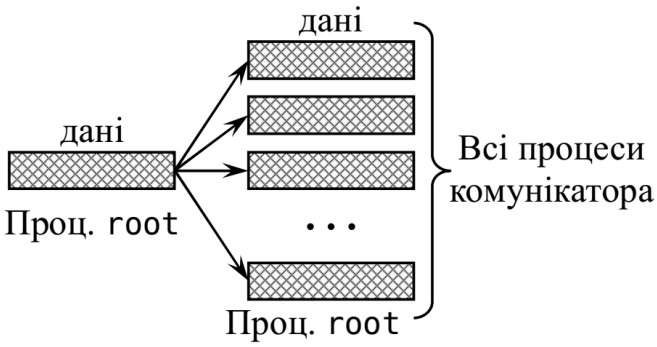


Рисунок 5.2 – Ілюстрація роботи методу `Bcast`

Зовні це виглядає так ніби-то один кореневий процес в циклі відправляє свій буфер даних всім процесам, а вони приймають його, приблизно так, як це показане у фрагменті нижче. Але насправді ніякого циклу не існує, а відправлення і прийом виконуються одночасно.

```
if (rank == root)
  for (int i = 0; i < size; i++) {
    if (i == root)
      recvBuf = Arrays.copyOf(sendBuf, sendBuf.length);
    else
      MPI.COMM_WORLD.Send(sendBuf, 0, sendBuf.length, MPI.INT,
                          i, 3);
  }
else
  MPI.COMM_WORLD.Recv(recvBuf, 0, recvBuf.length, MPI.INT,
                      root, 3);
```

Наступний колективний метод, який відповідає процедурі `MPI_GATHER` і має оголошення:

```
public void Gather(Object sendbuf, int sendoffset, int sendcount,
                   Datatype sendtype, Object recvbuf,
                   int recvoffset, int recvcount,
                   Datatype recvtype, int root)
                                        throws MPIException
```

збирає до одного масиву у буфері `recvbuf`, який належить процесу з рангом `root` і має розмір не менш за `sendcount*size` елементів, `sendcount` елементів даних типа `sendtype` з масивів `sendbuf` всіх процесів комунікатора (включаючи і процес `root`). Тобто кожен процес, в тому числі і сам `root`, посилає вміст свого буфера `sendbuf` починаючи з позиції `sendoffset` до процесу `root`, який збирає та зберігає дані в буфері `recvbuf`, починаючи з позиції `recvoffset` і розташовуючи їх у порядку зростання номерів процесів.

При виклику методу для виділеного процесу с рангом `root` істотними є значення всіх параметрів, а для інших процесів − тільки значення параметрів

sendbuf, sendoffset, sendcount, sendtype і root. Значення параметру root повинне бути однаковим у всіх процесів. Параметр recvcount у процесі root означає не сумарне число елементів типа recvtype, прийнятих від всіх процесів, а лише кількість даних, які приймаються від кожного окремого процесу. Типи даних, які передаються sendtype і приймаються recvtype повинні бути однакові.

Наприклад, для того щоб процес з рангом 2 зібрав у масив recvBuf по 10 цілочисельних елементів з масивів sendBuf процесів у комунікаторі COMM_WORLD застосування, потрібно, щоб у всіх процесах зустрівся наступний виклик методу Gather:

```
MPI.COMM_WORLD.Gather(sendBuf, 0, 10, MPI.INT, recvBuf, 0, 10,
                      MPI.INT, 2);
```

Дію методу Gather можна умовно проілюструвати наступною схемою (рис. 5.3).

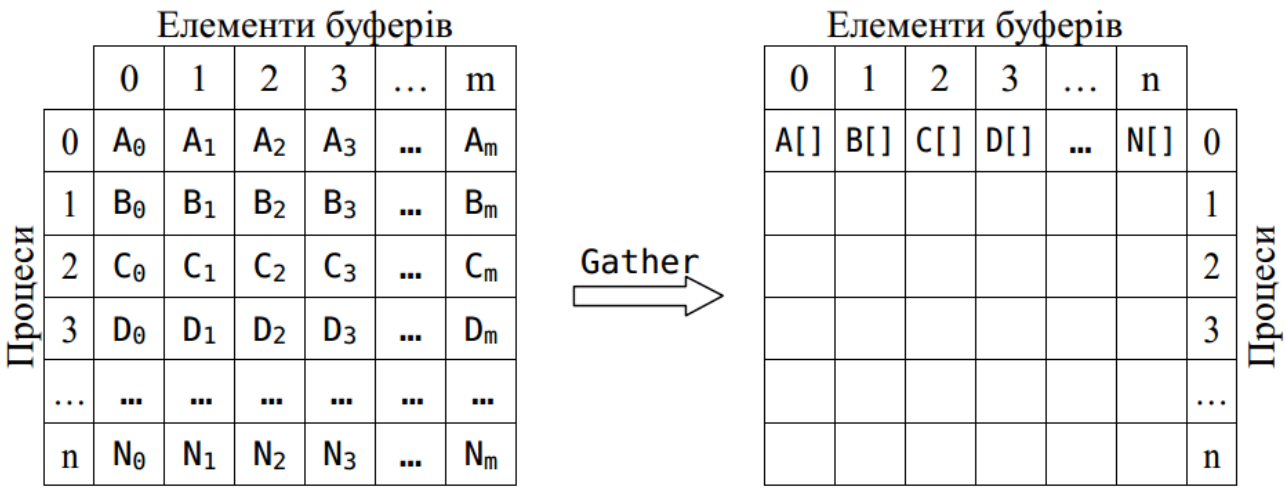


Рисунок 5.3 – Схема збору даних методом Gather

Інакше кажучи, дані з буферів всіх процесів комунікатора послідовно, в залежності від рангу процесу, від якого вони надійшли, розташовуються у буфері одного виділеного процесу. Розмір такого буфера мінімально дорівнює сумі розмірів всіх буферів, з яких збираються дані. Кількість даних, що збираються

від процесів повинна бути однаковою. Виконання методу Gather ілюструється на рис. 5.4

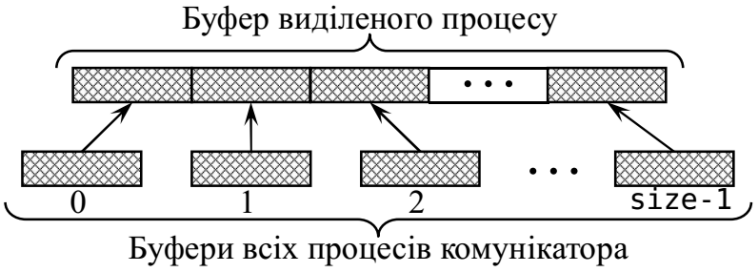


Рисунок 5.4 – Ілюстрація виконання методу Gather

Застосовуючи методи комунікації типа точка-точка, дію метода Gather можна імітувати таким фрагментом:

```
int[] message = new int[len], gather = new int[len * size];
if (myrank != root)
  MPI.COMM_WORLD.Send(message, 0, len, MPI.INT, root, myrank);
else
  for(int i = 0; i < size; i++)
    {
      if (i == root) {
        k = root * len;
/* Тут виконується підготовка власного масиву message для
   перенесення у відповідне місце масиву gather */
      }
      else {
        Status status = MPI.COMM_WORLD.Recv(message, 0, len,
                                            MPI.INT, i, MPI.ANY_TAG);
        k = status.source * len;
      }
      for(int j = 0; j < len; j++) gather[k + j] = message[j];
    }
```

Стандартні бібліотеки MPI містять ще й процедуру з ідентифікатором MPI_GATHERV, яка подібна до описаної вище, але в дечому відрізняється від неї. Тому й у пакеті MPJ Express оголошений її метод-аналог Gatherv:

```
public void Gatherv(Object sendbuf, int sendoffset,
                    int sendcount, Datatype sendtype,
                    Object recvbuf, int recvoffset,
                    int[] recvcount, int[] displs,
                    Datatype recvtype, int root)
                                        throws MPIException
```

Метод розширює функціональність методу `Gather` таким чином, що дозволяє збирати різну кількість даних від різних процесів комуникатору. Цей метод також здійснює збір даних з масивів `sendbuf`. Але місце розташування даних у результуючому буфері `recvbuf` вказується у допоміжному цілочисельному масиві `displs`, цей цілочисельний масив містить додаткове зміщення відносно початкового зсуву `recvoffset` у масиві `recvbuf` (тобто для повної відповідності процедурі `MPI_GATHERV` бажано встановлювати значення `recvoffset` рівним нулю). Розмір масиву `displs` дорівнює числу процесів у комунікаторі, а індекси елементів масиву відповідають рангам процесів, які посилають дані. Тобто дані, послані процесом із рангом `j`, розміщаються в `j`-му блоці буфера `recvbuf` на процесі `root`, що починається зі зміщенням, яке відповідає значенню в `displs[j]`. Ще один цілочисельний масив − `recvcount` містить кількість елементів, які передаються від кожного процесу. Індекс елемента масиву дорівнює рангу процесу, який посилає дані, а розмір масиву дорівнює числу процесів у комунікаторі.

Таким чином при використанні цього методу, дані, що збираються у буфері виділеного процесу, можуть мати різну кількість елементів (визначається вмістом масиву `recvcount`) та й розташовуватись у буфері вже можуть не так щільно, як у попередньому випадку, а на деякій відстані одне від одного (визначається вмістом масиву `displs`). Наочно це ілюструється на рис. 5.5.

Є цілком зрозумілим, що в *цьому разі розмір буфера виділеного процесу може перевершувати суму розмірів буферів всіх процесів комунікатора*.

Особливо слід підкреслити, що значення елементів масиву `displs` повинні бути такими, *щоб не відбувалося перекриття пакетів, які розміщуються у буфері головного процесу*, але між ними можуть бути «пусті дірки», як це зо-

бражено на рис. 5.5, або вони можуть розташовуватися щільно так, як при виконанні методу Gather.

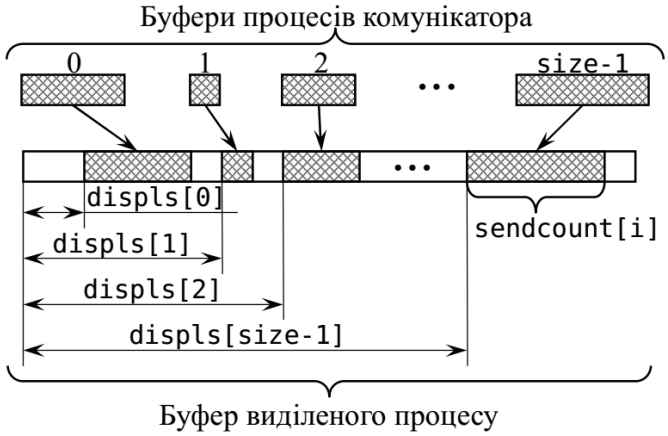


Рисунок 5.5 – Ілюстрація збирання даних методом Gatherv

Клас Intracomm містить ще два методи які є аналогами стандартних процедур MPI_SCATTER й MPI_SCATTERV, дія яких протилежна описаним вище. Це методи Scatter та Scatterv.

Перший з методів має опис:

```
public void Scatter(Object sendbuf, int sendoffset,
                    int sendcount, Datatype sendtype,
                    Object recvbuf, int recvoffset,
                    int recvcount, Datatype recvtype, int root)
                                        throws MPIException
```

На відміну від методу Gather цей *метод здійснює* не збір інформації від всіх процесів, а, навпаки, *розсилання по* sendcount *елементів даних типа* sendtype *з масиву* sendbuf, *що міститься у процесі* root *до масивів* recvbuf *всіх процесів* деякого *комунікатора*, *включаючи сам процес* root. *Можна вважати*, *що масив* sendbuf *поділяється на рівні частини за числом процесів*, кожна з яких складається з sendcount елементів типа sendtype, після чого j-та частина пересилається до j-го процесу.

Як і раніш, *для процесу* root *значимими є всі параметри*, *а для всіх інших процесів – тільки значення параметрів* recvbuf, recvoffset, recvcount,

recvtype і root. Значення параметру root повинне бути однаковим у всіх процесах. Однаковими повинні бути й типи даних, які розсилаються sendtype і даних, що збираються recvtype.

Ілюстрація дії методу Scatter може бути здійснена за допомогою наступної схеми (рис. 5.6), або за допомогою рис. 5.7.

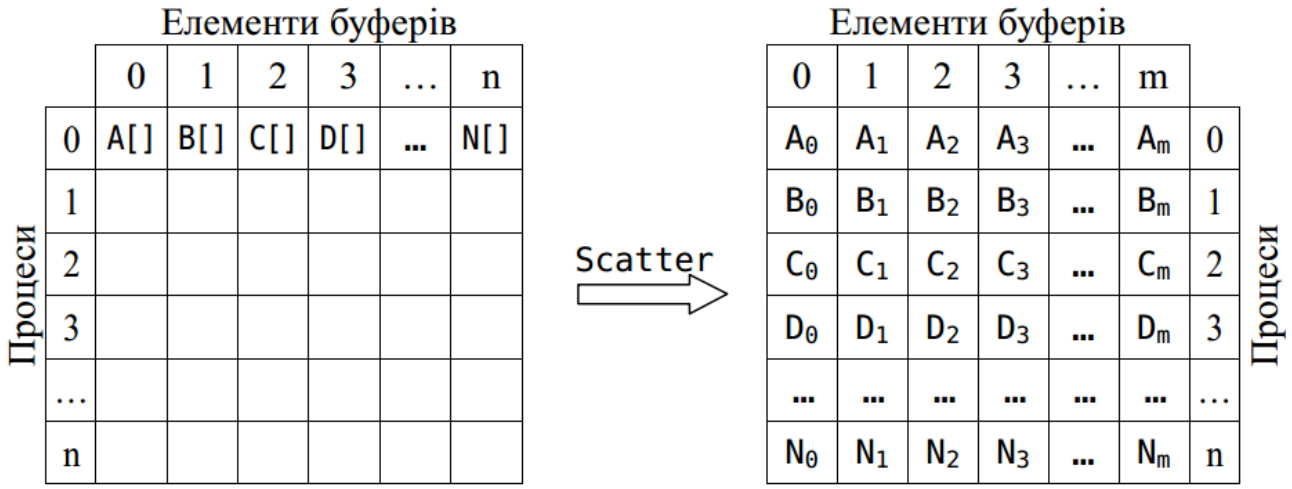


Рисунок 5.6 – Схема розподілення даних методом Scatter

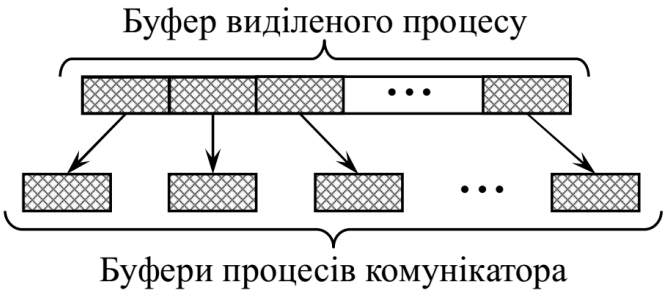


Рисунок 5.7 − Розподіл даних з великого буфера виділеного процесу рівними частками по маленьким буферам всіх процесів комунікатора

Автор навчального посібника [13] наводить приклад програми мовою C, яка демонструє застосування метода Scatter. У цьому прикладі у комунікаторі COMM_WORLD, який містить чотири процеси, останній процес source з рангом numtasks-1 виконує порядкове розсилання чисел типа float, що містяться у квадратній матриці sendBuf розмірністю 4 × 4, всім процесам застосування. У результаті окремі рядки матриці розташовуються у масивах recvBuf кожного

процесу, включаючи й сам процес `source`.

Оригінальна програма прекрасно працює і всі процеси дійсно одержують один рядок вихідної матриці, номер такого рядка збігається з рангом процесу. Але дослівний переклад цієї програми мовою Java, будучи запущеним на виконання, видає на термінал наступне повідомлення:

```
MPJ Express (0.38) is started in the multicore configuration
...
Caused by: java.lang.ClassCastException: [[F cannot be cast to [F
   at mpi.Comm.isend(Comm.java:827)
   at mpi.Intracomm.Scatter(Intracomm.java:886)
   at Scat.main(Scat.java:29)
   6 more
```

Тобто вкидається виняток, який говорить про те, що при виконанні метода `Scatter` не можливо провести перетворення типа двомірного масиву до типа одномірного. Таке явище пов'язане з тим, що оголошення двомірного масиву і привласнення значень його елементам в стилі мови Java:

```
float [][] = { {1.0, 2.0, 3.0, 4.0},
               {5.0, 6.0, 7.0, 8.0},
               {9.0, 10.0, 11.0, 12.0},
               {13.0, 14.0, 15.0, 16.0} };
```

не повністю відповідає оголошенню саме того ж масиву в стилі C:

```
float [SIZE][SIZE] = { {1.0, 2.0, 3.0, 4.0},
                       {5.0, 6.0, 7.0, 8.0},
                       {9.0, 10.0, 11.0, 12.0},
                       {13.0, 14.0, 15.0, 16.0} };
```

Різниця полягає в тому, що пам'ять, яка потрібна для розташування таким чином оголошеного масиву, компілятором мови C виділяється під час компіляції програми і елементи масиву розташовуються в ній послідовно один за одним: спочатку елементи першого рядка, потім одразу за ними елементи другого ряд-

ка, потім третього і так далі для всіх рядків масиву. Таке розташування елементів двомірного масиву в пам'яті повністю співпадає з розташуванням елементів одномірного масиву довжиною `SIZE` × `SIZE`, тому і програмисти і власне програма можуть розглядати відповідну послідовність комірок пам'яті так, як їм зручніше у даний відрізок часу, тобто або як двомірний масив, або як одномірний.

У програмі ж мовою Java, пам'ять під масив виділяється не статично під час компіляції програми, а динамічно під час її виконання. Тому хоча й елементи кожного рядка розташовуються так саме послідовно один за одним, але рядки можуть бути розташовані у різних, навіть несуміжних одна до одної ділянках пам'яті. Навіть у разі коли рядки теж послідовно розташовані друг за другом і створюється враження розміщення у пам'яті одномірного а не двомірного масиву, сувора типизованість мови Java не дозволяє подібної вільності в трактуванні даних програми. Якщо масив оголошений як двомірний, то він є тільки двомірним і вже не може розглядатись як одномірний. Тому й дослівно перекладена мовою Java програма вкидає виняток о неможливості перетворення типа двомірного масиву до одномірного.

Нижче наведений дещо змінений варіант перекладу програми (Ex_14) до якого добавлений метод програмного перетворення двомірного масиву до одномірного – `convertToOneDimention`, який просто переносить рядки матриці з масиву `sendMatr` до масиву `sendBuf`, розташовуючи них в ньому послідовно, подальші операції програми вже виконуються з цім масивом. В іншому програма повністю відповідає вихідній програмі, навіть збережена структура і стиль написання програми, саме так як мовою С.

```
import mpi.*;
import java.util.*;
class Ex_14 {
  private static float[] convertToOneDimention(float[][] matr,
                                               int size){
    float[] tmp = new float[size * size];
    for(int i = 0 ; i < size; i++)
```

```
      System.arraycopy(matr[i], 0, tmp, i * size, size);
    return tmp;
  }
  public static void main(String[] args) {
    final int SIZE = 4;
    int numTasks, rank, sendCount, recvCount, source;
    float[] sendBuf = new float[SIZE * SIZE];
    float[] recvBuf = new float[SIZE];
    MPI.Init(args);
    rank = MPI.COMM_WORLD.Rank();
    numTasks = MPI.COMM_WORLD.Size();
    source = numTasks - 1;
    if (rank == source) {
      float[][] sendMatr = {{1.0f, 2.0f, 3.0f, 4.0f},
                            {5.0f, 6.0f, 7.0f, 8.0f},
                            {9.0f, 10.0f, 11.0f, 12.0f},
                            {13.0f, 14.0f, 15.0f, 16.0f}};
      sendBuf = convertToOneDimention(sendMatr, SIZE);
    }
    if (numTasks == SIZE) {
      sendCount = SIZE;
      recvCount = SIZE;
      MPI.COMM_WORLD.Scatter(sendBuf, 0, sendCount, MPI.FLOAT,
                             recvBuf, 0, recvCount, MPI.FLOAT,
                             source);
      System.out.printf(" %nПроцес = %d одержав: %s", rank,
                        Arrays.toString(recvBuf));
    }
    else
      if (rank == source)
        System.out.printf(" %nКількість процесів повинна" +
                          " дорівнювати %d.%n", SIZE);
    MPI.Finalize();
   if (rank == source) System.out.printf("%n %n");
  }
}
```

Результат виконання перекладеної програми наведений нижче.

```
MPJ Express (0.38) is started in the multicore configuration

Процес = 3 одержав: [13.0, 14.0, 15.0, 16.0]
```

```
Процес = 0 одержав: [1.0, 2.0, 3.0, 4.0]
Процес = 2 одержав: [9.0, 10.0, 11.0, 12.0]
Процес = 1 одержав: [5.0, 6.0, 7.0, 8.0]
```

Розширення методу `Scatter` – метод `Scatterv` має таке оголошення:

```
public void Scatterv(Object sendbuf, int sendoffset,
                     int[] sendcount, int[] displs,
                     Datatype sendtype, Object recvbuf,
                     int recvoffset, int recvcount,
                     Datatype recvtype, int root)
                                        throws MPIException
```

Метод дозволяє виконувати розсилання різної кількості даних з масиву `sendbuf` (див. рис. 5.8). Початок порцій даних, що розсилаються, задається значеннями елементів масиву `displs[j]` відносно початкового зміщення `sendoffset`. Довжина масиву `displs` повинна дорівнювати числу процесів у поточному комунікаторі, а індекси елементів відповідають рангам процесів-адресатів.

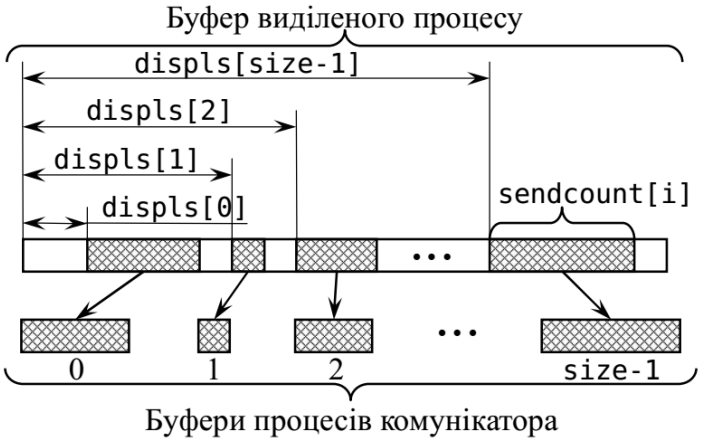


Рисунок 5.8 – Ілюстрація дії метода `Scatterv`

Кількість елементів, що передаються до кожного процесу, зберігається у значеннях елементів числового. Відповідно так само як і для масиву `displs`, індекси елементів у масиві `sendcount` дорівнюють рангам адресатів, а довжина масиву залається кількістю процесів у комунікаторі.

Дані, які посилаються від процесу root процесу с рангом j, розміщуються в j-му блоці буфера sendbuf, що починається зі зміщенням в displs[j] елементів типа sendtype відносно до початкового зміщення sendoffset у буфері sendbuf.

На відміну від нерегулярного метода збирання даних – Gatherv метод нерегулярного розподілу даних – Scatterv може від кореневого процесу відправляти до інших процесів пакети, *дані в яких можуть бути взятими з таких частин буферу, які частково або, навіть, повністю перекриваються між собою*.

Наступний метод-член класу Intracomm – Allgather є аналогом процедури MPI_ALLGATHER і дозволяє зібрати дані з буферів відправки даних *всіх процесів комунікатора до буферів прийому знову ж таки всіх його процесів*. Метод оголошується таким чином:

```
public void Allgather(Object sendbuf, int sendoffset,
                      int sendcount, Datatype sendtype,
                      Object recvbuf, int recvoffset,
                      int recvcount, Datatype recvtype)
                                        throws MPIException
```

Тобто іншими словами, цей метод ініціює обмін даними між всіма процесами. Дані збираються і зберігаються в порядку зростання номерів процесів. Блок даних, посланий процесом з рангом j, розміщується в j-му блоці буфера recvbuf приймаючого процесу. Операцію можна розглядати як метод Gather, при якому результат досягається на всіх процесах деякого комунікатора.

Наступна схема (рис. 5.9) ілюструє дію методу Allgather.

Імітацію дії методу Allgather наочно можна представити, наприклад, за допомогою такого простого коду:

```
for(int i = 0; i < size; i ++)
    MPI.COMM_WORLD.Gather(sendBuf, 0, sendBuf.length, MPI.INT,
                          recvBuf, 0, sendBuf.length, MPI.INT, i);
```

або можна запропонувати інший спосіб імітування без використання циклів:

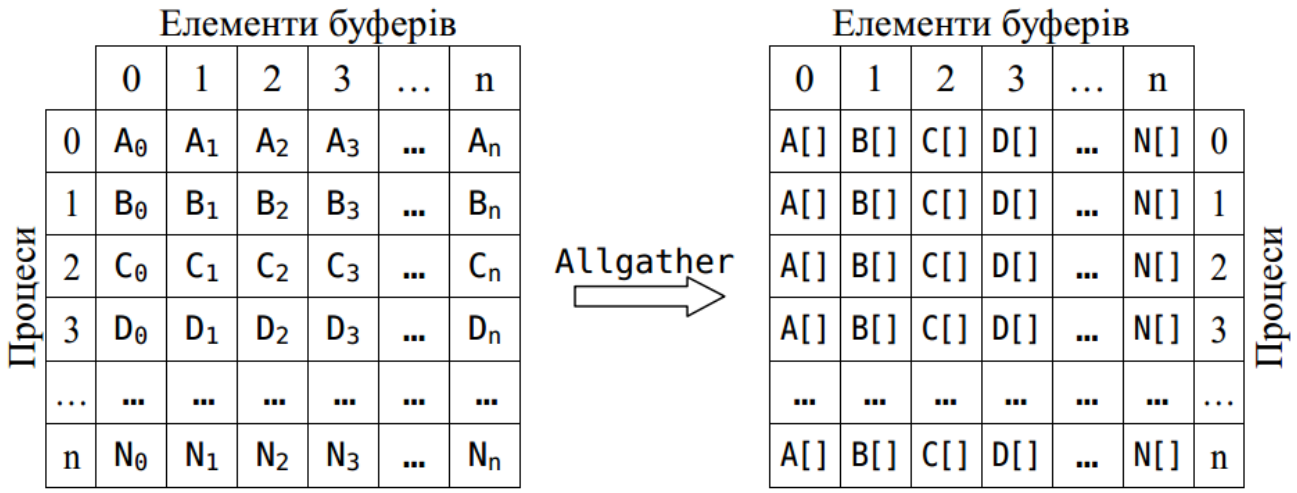


Рисунок 5.9 – Схема дії методу `Allgather`

```
MPI.COMM_WORLD.Gather(sendBuf, 0, sendBuf.length, MPI.INT,
                      recvBuf, 0, sendBuf.length, MPI.INT, root);
MPI.COMM_WORLD.Bcast(recvBuf, 0, recvBuf.length, MPI.INT, root);
```

І в тому й в іншому випадку результат буде саме такий, як і при безпосередньому використанні методу `Allgather`, але час витрачений на обидві імітації буде більшим.

Метод `Allgather` також має свого двійника, який поширює його функціональність. Це аналог процедури `MPI_ALLGATHERV` – метод на ім'я `Allgatherv`, який оголошується так:

```
public void Allgatherv(Object sendbuf, int sendoffset,
                       int sendcount, Datatype sendtype,
                       Object recvbuf, int recvoffset,
                       int[] recvcount, int[] displs,
                       Datatype recvtype)
                                          throws MPIException
```

Метод, як і попередні подібні до нього, виконує зборку різної кількості даних з масивів `sendbuf` від всіх процесів комунікатора. Порядок розташування даних у масиві `recvbuf` задається вмістом масиву `displs`, а кількість даних, одержуваних від процесів, заноситься у елементи масиву `recvcount`. *Для мето-*

ду справедливо теж зауваження, що й для методу `Gatherv`, *про неможливість перекриття даних у буферах прийому повідомлень*.

Наступна двійка колективних методів також здійснює обмін даними між процесами у комуникаторі, але, на відміну від попередніх, це вже *дійсно повний загальній обмін даними між усіма процесами і всіма буферами в них*. Методі, як і завжди, є аналогами процедур `MPI_ALLTOALL` і `MPI_ALLTOALLV`.

Перший з методів на ім'я `Alltoall` має наступне оголошення:

```
public void Alltoall(Object sendbuf, int sendoffset,
                     int sendcount, Datatype sendtype,
                     Object recvbuf, int recvoffset,
                     int recvcount, Datatype recvtype)
                                        throws MPIException
```

і дозволяє кожному процесу комунікатора *розсилати однакові за розміром порції даних до всіх інших процесів*, причому так, що `j`-й блок даних з буфера `sendbuf` `i`-го процесу попадають до `i`-го блоку даних у буфері `recvbuf` `j`-го процесу.

Наступна схема (рис. 5.10) може допомогти в розумінні, щодо механізму дії методу `Alltoall`.

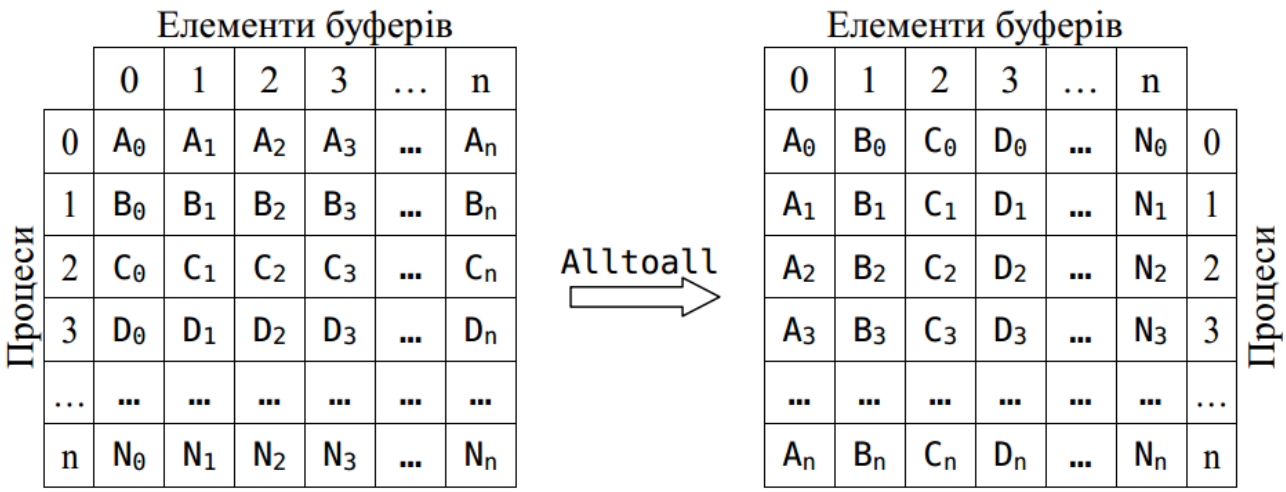


Рисунок 5.10 – Схема пересилання даних методом `Alltoall`

Якщо уявити собі вихідні дані до застосування методу `Alltoall` як деяку матрицю, яка по рядках розподілена по процесах комунікатора, то можна казати, що результатом виконання методу є транспонована матриця, рядки якої теж розподілені по відповідних процесах застосування.

І другий метод, знов таки розширення попереднього – `Alltoallv` з наступним оголошенням:

```
public void Alltoallv(Object sendbuf, int sendoffset,
                      int[] sendcount, int[] sdispls,
                      Datatype sendtype, Object recvbuf,
                      int recvoffset, int[] recvcount,
                      int[] rdispls, Datatype recvtype)
                                        throws MPIException
```

надає можливість *розсилання різної кількості даних від всіх процесів* активного комунікатора *до всіх процесів* в ньому. Розміщення даних у буфері `sendbuf` процесу, що здійснює відправку даних, як і раніш, визначається умістом масиву `sdispls`, а розміщення даних у буфері `recvbuf` процесів одержувачів даних визначається масивом `rdispls`. Кількість даних, що посилаються й приймаються визначається умістом масивів `sendcount` і `recvcount` відповідно. При використанні методу *не слід забувати про неможливість перекриття даних* в буферах де відбувається їхнє збирання.

Окрему групу у цьому сімействі методів-членів класу `Intracomm` складають методи, яки *дають можливість виконувати не тільки пересилання даних*, *а й в той самий час застосовувати до них ті*, *чи інші арифметичні або логічні операції*. У це сімейство також доданий механізм, який дозволяє використовувати *навіть спеціально розроблені більш складні користувальницькі операції над всіма елементами масивів*, які пересилаються у комунікаторі.

Першим в групі (так би мовити, її засновником) можна вважати метод-аналог процедури `MPI_REDUCE`. Метод має ім'я `Reduce` і таке оголошення:

```
public void Reduce(Object sendbuf, int sendoffset,
                   Object recvbuf, int recvoffset, int count,
```

```
                    Datatype datatype, Op op, int root)
                                        throws MPIException
```

Метод *виконує* `count` (дорівнює кількості даних, що пересилаються від кожного процесу до процесу з рангом `root`) *незалежних глобальних операцій* `op` *над відповідними елементами масивів* `sendbuf`. Результат виконання операції `op` над `i`-ми елементами масивів `sendbuf` всіх процесів активного комунікатора одержується в `i`-му елементі масиву `recvbuf` виділеного процесу `root`.

Схема на рис. 5.11 пояснює дію методу `Reduce`.

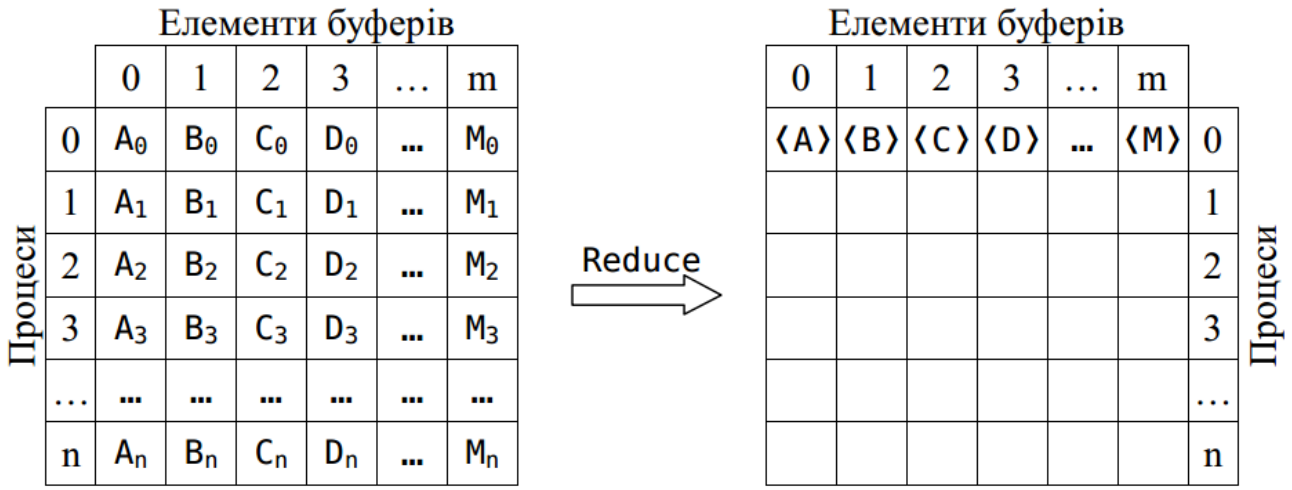


Рисунок 5.11 – Ілюстрація збору даних методом `Reduce` одночасно з виконанням деяких операцій над ними

Кутові дужки – `⟨⟩` на цієї схемі означають виконання деякої операції над елементами масиву, розподіленого по процесах, так, наприклад, `⟨D⟩` може означати обчислення глобальної суми або побітове I над елементами вектора `D`.

Відповідно до стандарту MPI в пакеті MPJ Express передбачена низка спеціальних статичних констант-членів класу MPI, за допомогою яких визначають глобальні операції `op`, що будуть виконуватись над даними. У табл. 5.2 наведені оголошення цих констант у відповідності з описами операцій, які виконуються, а також відповідні їм символи операцій у мові Java.

Таблиця 5.2 – Визначені константи операції у методі `Reduce`

Оголошення константи	Виконувана операція	Відповідне позначення у мові Java
`static Op PROD`	обчислення глобального добутку	*
`static Op SUM`	обчислення глобальної суми	+
`static Op BAND`	побітове І	&
`static Op BOR`	побітове АБО	|
`static Op BXOR`	побітове ВИКЛЮЧНЕ АБО	^
`static Op LAND`	логічне І (укорочене)	&&
`static Op LOR`	логічне АБО (укорочене)	||
`static Op LXOR`	логічне ВИКЛЮЧНЕ АБО	^
`static Op MAX`	визначення максимального значення в масиві	
`static Op MAXLOC`	визначення максимального значення і його позиції в масиві	
`static Op MIN`	визначення мінімального значення в масиві	
`static Op MINLOC`	визначення мінімального значення і його позиції в масиві	

Наведений нижче приклад (Ex_15) виконує глобальне підсумовування даних у великому масиві за допомогою схеми здвоювання з використанням пересилань даних типа точка-точка. Ефективність такого способу глобального підсумовування порівнюється з використанням колективного методу `Reduce` для досягнення того самого результату.

```
import mpi.*;
class Ex_15 {
  private int size, nProc, n;
  private static int rank, root = 0;
  private double[] a, b, c;
  private static String[] arg;
  public Ex_15(String[] args) {
    arg = MPI.Init(args);
    rank = MPI.COMM_WORLD.Rank();
    size = MPI.COMM_WORLD.Size();
    if (!((size > 1) && ((size & (~size + 1)) == size)))
      throw new IllegalArgumentException();
    n = Integer.parseInt(arg[0]);
```

```
  a = new double[n];
  b = new double[n];
  c = new double[n];
  nProc = size;
  Arrays.fill(a, 1.0 / size);
  c = Arrays.copyOf(a, a.length);
}
private void makeSum() {
  double time_start, time_finish;
  MPI.COMM_WORLD.Barrier();
  time_start = System.currentTimeMillis();
  while (nProc > 1) {
    if (rank < nProc / 2) {
      Status status = MPI.COMM_WORLD.Recv(b, 0, n, MPI.DOUBLE,
                                          nProc - rank - 1, 1);
      for(int i = 0; i < n; i++) c[i] += b[i]; }
    else if (rank < nProc)
          MPI.COMM_WORLD.Send(c, 0, n, MPI.DOUBLE,
                              nProc - rank - 1, 1);
    nProc /= 2; }
  b = Arrays.copyOf(c, c.length);
  time_finish = System.currentTimeMillis() - time_start;
  if (rank == root)
    System.out.printf(" %nПісля моделювання b[0] = %f", b[0]);
  System.out.printf(" %nПроцес = %d, час моделювання =" +
                    " %6.3f мс", rank, time_finish);
  Arrays.fill(a, 1.0 / size);
  MPI.COMM_WORLD.Barrier();
  time_start = System.currentTimeMillis();
  MPI.COMM_WORLD.Reduce(a, 0, b, 0, n, MPI.DOUBLE, MPI.SUM,
                        root);
  time_finish = System.currentTimeMillis() - time_start;
  if (rank == root)
    System.out.printf(" %nПісля виконання Reduce() b[0] = %f",
                      b[0]);
  System.out.printf(" %nПроцес = %d, час виконання Reduce()" +
                    " = %6.3f мс", rank, time_finish);
}
public static void main(String[] args) throws MPIException {
  try {
    Ex_15 ex = new Ex_15(args);
    ex.makeSum();
  }
```

```
      catch(ArrayIndexOutOfBoundsException e) {
        if (rank == root)
          System.err.printf(" %nНе вистачає параметра для" +
                            " методу main.%nСинтаксис команди" +
                            " запуску програми:%n %n\t\t" +
                            "mpjrun.sh -np <К-ТЬ_ПРОЦ> Ex_15" +
                            " <РОМІР_МАСИВУ>");
      }
      catch(NumberFormatException e) {
        if (rank == root)
          System.err.printf(" %nПараметр програми повинен бути" +
                            " числом.%nНаприклад, 100, а не" +
                            " \"%s\".", arg[0]);
      }
      catch(IllegalArgumentException e) {
        if (rank == root)
          System.err.printf(" %nКількість процесів повинна бути" +
                            " ступенем двійки.%nТобто 2, 4, 8," +
                            " 16, ..., а не \"%s\".", args[1]);
      }
      finally {
        MPI.Finalize();
        if (rank == root) System.err.printf("%n %n");
      }
    }
  }
```

Більшість стандартних операцій, які передбачені, як безпосередньо у стандарті MPI, так і в пакети класів MPJ Express особливих пояснень не потребують, оскільки є доволі очевидними. Але є сенс окремо зупинитись на двох операціях з табл. 5.2. Це операції `MPI.MINLOC` і `MPI.MAXLOC`.

Операція `MPI.MINLOC` застосовується для обчислення глобального мінімуму і індексу масиву, який йому відповідає. Так саме `MPI.MAXLOC` обчислює глобальний максимум і його індекс. Одним з використань методу може бути визначення глобального мінімуму (максимуму) і рангу процесора, який містить це значення. Операція `MPI.MAXLOC` визначається наступним чином:

$$\begin{pmatrix} u \\ i \end{pmatrix} \circ \begin{pmatrix} v \\ j \end{pmatrix} = \begin{pmatrix} w \\ k \end{pmatrix},$$

де – $w = \max(u, v)$, а $k = \begin{cases} i & \text{якщо } u > v \\ \min(i, j) & \text{якщо } u = v \\ j & \text{якщо } u < v \end{cases}$.

Аналогічно визначається і операція `MPI.MINLOC`:

$$\begin{pmatrix} u \\ i \end{pmatrix} \circ \begin{pmatrix} v \\ j \end{pmatrix} = \begin{pmatrix} w \\ k \end{pmatrix},$$

де – $w = \min(u, v)$, а $k = \begin{cases} i & \text{якщо } u < v \\ \min(i, j) & \text{якщо } u = v \\ j & \text{якщо } u > v \end{cases}$.

Обидві операції мають властивості асоціативності і комутативності. Слід зауважити, що якщо `MPI.MAXLOC` застосовується для скорочення довжини послідовності пар чисел $(u_0, 0)$, $(u_1, 1)$, …, $(u_{n-1}, n - 1)$, тоді повертається пара чисел (u, r), де $u = \max_i u_i$ та r є індексом першого глобального максимуму в послідовності. Таким чином, якщо кожен процес містить деяке значення і свій ранг у групі, то використання методу `Reduce` с операцією `Op = MPI.MAXLOC` поверне максимальне значення і ранг першого ж процесу, який містить таке значення. Аналогічно, `MPI.MINLOC` може бути використана щоб повернути мінімальне значення і його індекс. У загальному ж випадку, `MPI.MINLOC` обчислює, так званий, ***лексикографический мінімум,*** коли елементи упорядковуються відповідно до першого компонента кожної пари, а зв'язки вирішуються згідно з другим компонентом.

Операція редукції визначена для дій над аргументами, які складаються з пар: значення і індекс. Так саме як для Fortran і C, MPJ Expres для мови Java вводить тип даних, який описує таку пару. Слід відзначити, що в даному випадку пара являє собою масив даних базового типа, об'єднаних у окремий об'єкт добре пристосований для операцій з обміну даними. Іншими словами, розроб-

ники пакета не створювали для цієї мети новий клас даних а запозичили механізм властивий стандартним бібліотекам MPI для мови Fortran.

В разі використання операцій MPI.MINLOC і MPI.MAXLOC в методі редукування до нього у якості аргументу слід передавати пару чисел – значення і індекс. Для цього MPJ Express надає відповідні наперед визначені типи даних (див. табл. 5.1). Операції MPI.MAXLOC і MPI.MINLOC можуть бути використані для всіх типів, які у своєму визначенні містять суфікс 2, наприклад, MPI.SHORT2 або MPI.DOUBLE2.

Наведені нижче фрагменти програм пояснюють сказане про використання операцій MPI.MAXLOC і MPI.MINLOC.

У першому фрагменті кожен з запущених процесів містить масив випадкових чисел ain[], розмір якого визначається змінною N. Всі процеси комунікатора, виконуючи метод Reduce, порівнюють i-ті елементи власного масиву з відповідними елементами в масивах інших процесів знаходять максимальне значення і поміщують його та свій ранг у результуючий масив out[], який розміщується у кореневому процесі root. Але перед тим, як звертатись до методу Reduce, необхідно підготувати його вхідний параметр – масив in[], розмір якого вдвічі більший за розмір вихідного масиву ain[]. Для цього масив ain[] поелементно переноситься у нульовий і у всі парні елементи масиву in[], а у непарні елементи масиву заноситься ранг відповідного процесу. Це треба зробити тому, що операція MAXLOC виконується одразу над парами: «число – ранг». Підготовлений таким чином масив передається у Reduce у якості першого вхідного параметра, тип даних в масиві вказується як DOUBLE2. Одержаний у кореневому процесі результуючий масив out теж подвійної довжини, як і вхідний, розподіляються на два масиви aout[] і ind[], вміст яких й роздруковується. В першому з них містяться максимальні знайдені числа, а у другому ранги процесів, які ними володіють.

```
...
  MPI.Init(args);
  int N = 10000;
```

```
    double[] ain = new double[N], aout = new double[N];
    int[] ind = new int[N];
    double[] in = new double[N * 2], out = new double[N * 2];
    int i, myrank, root = 0;
    Random rnd = new Random();
    myrank = MPI.COMM_WORLD.Rank();
    for (i = 0; i < N; i++) ain[i] = rnd.nextDouble() * 10;
    for (i = 0; i < N * 2; i += 2) {
      in[i] = ain[i / 2];
      in[i + 1] = myrank;
    }
    MPI.COMM_WORLD.Reduce(in, 0, out, 0, N, MPI.DOUBLE2,
                          MPI.MAXLOC, root);
  // В цій точці коду результат виконання орерації розташований
  // у масиві out процесу root
  // Далі виконується розпакування і друк остаточних результатів
    if (myrank == root) {
      for (i = 0; i < N; i++) {
        aout[i] = out[i * 2];
        ind[i] = (int)out[i * 2 + 1];
      }
      System.out.printf(" %n%s%n%s%n %n", Arrays.toString(aout),
                        Arrays.toString(ind));
    }
  ...
```

Інший фрагмент коду демонструє знаходження глобального мінімуму, його індексу у масиві і номера процесу до якого він належить. Пошук ведеться у великому масиві, якимсь чином (наприклад, за допомогою метода `Scatter` або пари методів `Send/Recv`) розподіленому по процесах комунікатора. У фрагменті, спочатку кожен процес звичайним образом шукає локальний мінімум у власній частині масиву, а потім з метою знаходження глобального мінімуму виконується звертання до методу `Reduce`, якому передається параметр `in[]` з двох елементів. В перший елемент масиву уміщується попередньо знайдений процесом локальний мінімум, а потім розраховується і у другий елемент заноситься індекс, який є індексом мінімуму у глобальному масиві. Метод `Reduce` виконує операцію `MINLOC` над сформованими таким чином парами чисел, у якості типа оброблюваних даних використовується тип `MPI.FLOAT2`. Знайдений глобальний

мінімум і його індекс у вихідному масиві повертається у елементах масиву `out[]`. Наприкінці можна розрахувати ранг процесу, який містить глобальний мінімум і його індекс у локальній частині масиву.

```
...
public static void main(String[] args) {
  int LEN = 100000;             // Розмір вихідного масиву даних
  float[] sVal = new float[LEN], // Глобальний масив значень
          in = new float[2], out = new float[2];
  int myRank, commSize, minRank, globalIndex, localIndex,
      root = 0;
  float minVal;
  MPI.Init(args);
  myRank = MPI.COMM_WORLD.Rank();
  commSize = MPI.COMM_WORLD.Size();
  int count = LEN / commSize; // Кількість даних в локальних
                              // масивах на процесах
  float[] rVal = new float[count];
  if (myRank == root) {
    Random rnd = new Random();
    for ( int i = 0; i < LEN; i++)
      sVal[i] = rnd.nextFloat() * 100;
  }
/* Тут виконується розсилання великого масиву по процесах */
// Далі пошук локального мінімуму всіма процесами комунікатора
  in[0] = rVal[0];
  for (int i = 1; i < count; i++)
    if (in[0] > rVal[i]) {
      in[0] = rVal[i];
      in[1] = i;
    }
// Пошук глобального мінімуму серед знайдених локальних
  in[1] = myRank * count + in[1];
  MPI.COMM_WORLD.Reduce(in, 0, out, 0, 1, MPI.FLOAT2,
                        MPI.MINLOC, root);
// Результат пошуку у масиві out[] на процесі root
  if (myRank == root) {
    minVal = out[0];
    globalIndex = (int)out[1];
    minRank = (int)out[1] / count;
    localIndex = (int)out[1] % count;
// Тут можна роздрукувати отримані результати
```

```
    }
  }
  ...
```

Є цілком очевидним та й взагалі зрозумілим, що передбачених в пакеті у кількості вісьмох (відповідають операціям мови Java) стандартних глобальних операцій не достатньо для вирішення широкого кола задач, які виникають перед програмістами. Тому стандарт MPI передбачає можливість для програмістів створювати свої власні методи для виконання глобальних операцій, які можна застосовувати сумісно з методом `Reduce`.

Такий механізм підтримується й в пакеті MPJ. Однак створення власного методу глобальної операції в мові Java у дечому істотно відрізняється від механізму, що застосовується умовах Fortran і C. Концепція мови, яка прийнята її розробниками, не надає можливості оголосити окремий зовнішній метод, який би не належав до якогось класу, а навпаки був би окремим зовнішнім методом, як це має місце для функцій і підпрограм у інших мовах. У Java механізм подібний до зовнішніх функцій моделюється за допомогою методів внутрішніх класів. З метою здійснення такого моделювання, до пакету MPJ додані два спеціальних класи, призначенням яких власне є надання механізму створення власної глобальної операції.

По-перше – це абстрактний клас `User_function`, оголошений таким чином:

```
  public abstract class User_function {
    public User_function() {/* конструктор за умовчанням */};
    public abstract void Call(Object invec, int inoffset,
                              Object inoutvec, int inoutoffset,
                              int count, Datatype datatype)
                                         throws MPIException {}; }
```

І по-друге – це клас `Op`, інтерфейс якого має наступний вигляд:

```
  public class Op {
    public int opCode;
```

```
    public Op(User_function function, boolean commute)
                          throws MPIException;
    public Op(User_function function, boolean commute,
              int opCode_) throws MPIException;
    public void free() throws MPIException;
    public void finalize() throws MPIException;
  }
```

Як видно з оголошення методу `Reduce`, параметр глобальної операції `op`, який передається до нього, є посиланням на об'єкт типа `Op`. Тому, в загалі кажучи, для створення глобальної операції досить лише за допомогою одного з двох конструкторів класу `Op` (аналоги процедури `MPI_OP_CREATE`) створити об'єкт цього класу і тім самим одержати посилання на нього. Але для того щоб створити об'єкт `op` до конструктора класу `Op` необхідно у якості одного з параметрів передати посилання на об'єкт, який має тип `User_function`. Але через те, що клас `User_function` є абстрактним класом, не можна створити об'єкт безпосередньо такого типа, а необхідно породити від абстрактного класу похідний субклас в якому обов'язково перевизначити абстрактний метод `Call`. Цей метод фактично є аналогом зовнішньої функції, яка стандартно оголошується в MPI-програмі для створення власної операції над даними:

```
FUNCTION FUNC(INVEC(*), INOUTVEC(*), LEN, TYPE).
```

Методу `Call` навіть передаються ті ж самі параметри, що й у функцію `FUNC`. Першому аргументу операції, яка буде виконуватись, відповідає перший параметр методу – `invec`, другому аргументові − параметр `inoutvec`, а результат виконання операції мусить також повертатись у параметрі `inoutvec`. Параметр методу `count` визначає кількість елементів вхідного та вихідного масивів, а параметр `datatype` − тип вхідних і вихідних даних. На код користувальницького методу `Call` *накладається одне обов'язкове обмеження* – цей метод *не може робити* яких-небудь *викликів методів класів пакету* MPJ *для виконання обміну даними*.

Створювана таким чином *глобальна операція обов'язково повинна бути асоціативною*. А параметр булевого типа – `commute`, який передається до конструкторів `Op`, в разі коли його значення дорівнює `true` надає їй властивість бути також й *комутативною*. Якщо ж значення параметру `commute` дорівнює `false` – *порядок виконання глобальної операції строго фіксується відповідно до зростання рангів процесів*, починаючи з процесу з рангом 0.

Знищення користувальницької глобальної операції `op`, в разі її подальшої непотрібності, здійснюється за допомогою методу `free` класу `Op` – аналогу процедури `MPI_OP_FREE`, однак його використання не є обов'язковим тому, що дії по видаленню вже непотрібних об'єктів виконує збірник сміття віртуальної машини Java.

Наступний приклад програми (Ex_16) демонструє спосіб описання користувальницького методу для використання у якості глобальної операції. У прикладі оголошений внутрішній клас `My_function`, який розширює абстрактний клас `User_function` пакета MPJ, і в ньому перевизначається абстрактний метод `Call`. Перевизначення зроблено так, щоб отриманий метод обчислював поелементну суму за модулем п'ять для двох векторів `invec` і `inoutvec`, які містять цілі числа. Для здійснення тісного зв'язку проміж класами у зовнішньому класі `Ex_16`, що охоплює створений внутрішній клас, оголошено поле `sumMod5`, яке належить до типа `My_function`. Ініціалізація цього поля обов'язково повинна виконуватись у деякому методі зовнішнього класу. З метою такої ініціалізації й введений спеціальний метод зовнішнього класу `initMy_function`, у якому просто конструктором без параметрів і створюється об'єкт. На наступному кроці ініційоване посилання на цей об'єкт передається до конструктора спеціального класу пакета MPJ – `Op`. Такі дії практично є аналогом звернення до процедури `MPI_OP_CREATE` в звичайних бібліотеках MPI. Тут же просто створюється об'єкт `op`, який вже слугує параметром для визначення глобальної операції в методі `Reduce`. Після виконання необхідних розрахунків, наприкінці програми посилання на об'єкт глобальної операції видаляється за допомогою методу об'єкта `free` (аналог процедури `MPI_OP_FREE`). За ідеологією мови Java, виклик цього

методу можна не робити, збирач сміття віртуальної машини видалить посилання коли мінує необхідність в його використанні.

```
import mpi.*;
import java.util.*;
class Ex_16 {
  private class My_function extends User_function {
    public void Call(Object invec, int inoffset, Object inoutvec,
                     int inoutoffset, int count,
                     Datatype datatype) throws MPIException {
      int[] a = new int[count], b = new int[count];
      System.arraycopy(invec, inoffset, a, inoffset,
                       count * datatype.Size ());
      System.arraycopy(inoutvec, inoutoffset, b, inoutoffset,
                       count * datatype.Size());
      for(int i = inoffset; i < count; i++)
        b[i] = (a[i] + b[i]) % 5;  // Знаходження суми масиву a[]
                                   // за модулем 5. Всі b[i] = 0
      System.arraycopy(b, inoutoffset, inoutvec, inoutoffset,
                       count * datatype.Size());
    }
  }
  private My_function smod5;
  private static int rank, root = 0;
  private static String[] arg;
  private int size, n, a[], b[];
  private static Ex_16 ex;
  public Ex_16(String[] args) {
    arg = MPI.Init(args);
    rank = MPI.COMM_WORLD.Rank();
    size = MPI.COMM_WORLD.Size();
    n = Integer.parseInt(arg[0]);
    a = new int[n];
    b = new int[n];                // Зараз всі b[i] дорівнюють 0
    for(int i = 0; i < n; i++) a[i] = rank + i;
  }
  private void initMy_function() { smod5 = new My_function(); }
  private void printArray(String str, int ar[], int count) {
    System.out.printf(" %n%s%s...", str,
                      Arrays.toString(Arrays.copyOf(ar, count)));
  }
  public static void main(String[] args) throws MPIException {
```

```
    boolean canDo = true;
    try { ex = new Ex_16(args); }
    catch(ArrayIndexOutOfBoundsException e) {
      canDo = false;
      if (rank == root)
        System.err.printf(" %nНе вистачає параметра для" +
                          " методу main.%nСинтаксис команди" +
                          " запуску програми:%n %n\t\t" +
                          "mpjrun.sh -np <К-ТЬ_ПРОЦ> Ex_13" +
                          " <РОМІР_МАСИВУ>");
    }
    catch(NumberFormatException e) {
      canDo = false;
      if (rank == root)
        System.err.printf(" %nПараметр програми повинен бути" +
                          " числом.%nНаприклад, 7815, а не" +
                          " \"%s\".", arg[0]);
    }
    finally {
      if (canDo) {
        ex.initMy_function();          // Відповідає оголошенню
                                       // "external smod5"
        String str = "Процес " + ex.rank + ", масив a = ";
        ex.printArray(str, ex.a, 15);
        Op op = new Op(ex.smod5, true);  // Фактично означає
                 // "call MPI_OP_CREATE(smod5, .TRUE., op, ierr)"
        MPI.COMM_WORLD.Reduce(ex.a, 0, ex.b, 0, ex.n, MPI.INT,
                              op, ex.root);
        op.free();                 // Відповідає стандартному
                                   // "call MPI_OP_FREE(op, ierr)"
        if(ex.rank == ex.root) {
          System.out.printf("%n");
          ex.printArray("Результат у масиві b = ", ex.b, 10);
          System.out.printf("%n %n");
        }
      }
      MPI.Finalize();
    }
  }
}
```

В класі `Intracomm` є ще декілька методів саме цього сімейства.

Наприклад метод-аналог процедури MPI_ALLREDUCE з оголошенням:

```
public void Allreduce(Object sendbuf, int sendoffset,
                      Object recvbuf, int recvoffset,
                      int count, Datatype datatype, Op op)
                                        throws MPIException
```

є повністю аналогічним попередньо описаному методу Reduce. Відмінність методу Allreduce від свого попередника є лише у тому, що *результат виконання операції* op *отримується у масивах* recvbuf *всіх процесів*.

У наступному фрагменті коду кожен процес обчислює суми елементів у рядках локальних масивів a, після чого отримані суми зі всіх процесів підсумовуються за допомогою методу Allreduce і результат розташовується у масивах r на всіх процесах застосування.

```
for(i = 0; i < n; i++) s[i] = 0.0;
for(i = 0; i < n; i++)
  for(j = 0; j < m; j++) s[i] = s[i] + a[i][j];
MPI.COMM_WORLD.Allreduce(s, 0, r, 0, n, MPI.DOUBLE, MPI.SUM);
```

Ще один метод з тієї ж групи Reduce − аналог стандартної процедури MPI_REDUCE_SCATTER. Це метод з оголошенням:

```
public void Reduce_scatter(Object sendbuf, int sendoffset,
                           Object recvbuf, int recvoffset,
                           int[] recvcounts,
                           Datatype datatype, Op op)
                                        throws MPIException
```

Як видно з імені методу і його оголошення, метод призначений для одночасного виконання деякої кількості глобальних операцій op над відповідними елементами масивів sendbuf і розподілу результатів операцій по всіх процесах у комуніуаторі. Кількість виконуваних операцій визначається вмістом масиву recvcounts і у загалі дорівнює сумі всіх елементів масиву. Функціонально це еквівалентно тому, що спочатку виконуються глобальні операції, потім резуль-

тат розсилається по процесах, таким чином, що i-ий процес одержує i-у порцію результатів кількістю recvcounts[i] елементів і поміщає них до свого масиву recvbuf. *При виконанні операцій масив* recvcounts *на всіх процесах активного комунікатора повинен бути однаковим*.

За допомогою методу Scan – аналога процедури MPI_SCAN, який має оголошення:

```
public void Scan(Object sendbuf, int sendoffset,
                 Object recvbuf, int recvoffset,
                 int count, Datatype datatype, Op op)
                                        throws MPIException
```

виконується просто count незалежних часткових глобальних операцій op над відповідними елементами масивів sendbuf. Алгоритм роботи методу є таким, що i-ий процес виконує count глобальних операцій над відповідними елементами масивів sendbuf інших процесів з номерами від 0 і до i включно, а потім поміщає отриманий результат у масив recvbuf. Загальній повний результат таких часткових глобальних операції збирається у масиві recvbuf останнього з процесів.

5.4.1 Контрольні запитання та завдання для самостійної роботи

− В чому полягає основна відмінність колективних операцій обміну даними від операцій взаємодії типа точка-точка?

− Назвіть чотири категорії операцій колективної взаємодії.

− Можна або ні стверджувати, що в колективних взаємодіях одночасно беруть участь усі процеси застосування?

− Що гарантується стандартом MPI при поверненні з методів колективної взаємодії?

− Існують які-небудь обмеження з боку колективних операцій на виконання разом з ними ще й операцій точка-точка в тому самому комунікаторі?

– Що Ви можете сказати про синхронізацію у виконанні операцій колективної взаємодії?

– Яка особливість характерна для колективних операцій с точки зору атрибутів повідомлень, які пересилаються операціями?

– Розкажіть про дію і застосування методу `Barrier`.

– Дайте розгорнуту характеристику метода відправлення повідомлення `Bcast`.

– Можна або ні за допомогою методу `Recv` прийняти повідомлення, яке було відправлено методом `Bcast`?

– Охарактеризуйте методи `Gather` і `Gatherv` їхнє призначення. В чому полягає різниця між цими методами, як вона позначається на параметрах, які до них передаються.

– Які дії виконують методи `Scatter` і `Scatterv`. Спробуйте написати фрагмент коду з використанням операцій точка-точка, який може ілюструвати, наприклад, метод `Scatter`.

– Сформулюйте у чому полягає розходження у алгоритмах функціонування між методами `Bcast` і `Scatter`?

– Спробуйте переписати програму Ex_14 застосовуючи замість методу `Scatter` операції типа точка-точка. Порівняйте ефективність цих двох програм.

– Докладно опишіть метод `Allgather`. Охарактеризуйте інші методи (`Allgatherv`, `Alltoall` і `Alltoallv`) з першої групи колективних методів взаємодії процесів.

– Яке призначення методу редукування `Reduce`? Поясніть дію метода.

– Перечисліть операції, які можуть виконуватись на тлі метода, надайте їхні позначення у синтаксисі мови Java.

– Для знаходження яких параметрів даних, що розподілені по процесах, призначені операції редукування `MPI.MINLOC` і `MPI.MAXLOC`? Опишіть як вони виконуються, для яких типів даних (у пакеті MPJ Express) вони застосовуються?

– Опишіть класи і методи, які призначені для розширення кількості операцій, що можна виконувати в методах редукування. Розкажіть про принцип створення операцій користувача.

– Охарактеризуйте методи `Allreduce` та `Reduce_scatter`.

– Для чого призначений метод `Scan`? Опишіть його алгоритм роботи.

– За допомогою пересилань точка-точка змоделюйте бар'єрну синхронізацію (Ex_13) і порівняйте ефективність такої реалізації із виконанням стандартного методу `Barrier`.

– Відкомпілюйте і запустить програму Ex_15 і порівняйте ефективність такої реалізації з використанням стандартного методу `Reduce`.

– Напишіть свій варіант методу `Gather`, використовуючи методи пересилання повідомлень типа точка-точка, порівняйте ефективність цього і бібліотечного метода.

– Подумайте, як організувати колективний асинхронний обмін даними, який був би аналогічним до алгоритму роботи метода `Reduce`. Напишіть таку програму.

– Спробуйте написати програму, результат роботи якої був би аналогічним до методу `Alltoall`. Емуляцію метода виконати за допомогою асинхронних операцій пересилання даних.

– Якщо у Вас є можливість доступу до обчислювального кластера, дослідіть масштабованість (залежність часу виконання від числа процесів) різних колективних операцій пакету MPJ Express на Вашому конкретному обладнанні.

5.5 Операції з групами процесів і комунікаторами

Досі розглядалася взаємодія паралельних процесів у межах одного комунікатора. Однак, на практиці часто виникають ситуації, коли вирішити поставлену задачу в повному обсязі можна тільки забезпечивши створення нового незалежного комунікатора.

Наприклад, це буває необхідно, щоб дати можливість деякій групі процесів працювати над своєю незалежною підзадачею. Саме така ж потреба може

виникнути у разі, коли існує деяка особливість алгоритму, через яку тільки частина процесів повинна обмінюватися даними і буває зручним завести для їхньої взаємодії окремий комунікатор. Ще одним прикладом може бути те, що при створенні бібліотек підпрограм ще більш високого рівня потрібно гарантувати, щоб пересилання даних у бібліотечних модулях не перетинались з пересиланнями в основній програмі. Виходячи з цього, стандарт MPI вимагає включення до бібліотек спеціальних методів, які повинні надати широкі можливості для операцій над групами процесів і комунікаторами. Ці вимоги повністю задовольняються і в пакеті класів MPJ Express.

5.5.1 Операції із групами процесів

Крім поняття комунікатор стандарт MPI уводить ще й *поняття групи*. ***Група*** − це впорядкована множина процесів, виділена в середині якого-небудь комунікатора. Група може містити, як всі процеси комунікатора, так і набір з декількох окремих заздалегідь визначених процесів. Один і той же процес може одночасно входити у різні групи. Як і комунікатори, кожна група має свій унікальний ідентифікатор оскільки являє собою екземпляр об'єкта типа `Group`. В класі `MPI` пакету передбачені дві статичні константи для ідентифікації груп з особливими властивостями. По-перше це пуста група, *об'єкт якої вже ініційований, але яка у даний відрізок часу ще не містить жодного процесу*. Ідентифікатор такої групи визначений як `public static Group GROUP_EMPTY`. І по-друге це *група об'єкт якої* поки що навіть *не створений*. Ця, так би мовити, помилкова група ідентифікується ім'ям `public static Group GROUP_NULL`. Самі константи і те, як вони застосовуються у програмах, повністю відповідає застосуванню констант-аналогів `MPI_GROUP_EMPTY` і `MPI_GROUP_NULL` у стандарті MPI.

Кожному процесу в групі, так саме як і в комунікаторі, зіставлене ціле число − його ранг або номер. Ранги процесів в групах і у відповідних комунікаторах можуть збігатися, але можуть бути й різні.

Нові групи можна створювати, як на основі комунікаторів, так і на основі вже існуючих груп, але слід пам'ятати, що в операціях обміну між процесами можуть використовуватись тільки комунікатори. *Базова група*, *з якої створюються всі інші групи процесів*, *завжди пов'язана з комунікатором* `COMM_WORLD` (статичне поле-член класу `MPI`), до неї входять всі процеси застосування. Операції над групами процесів є повністю локальними, тобто участь у них може приймати тільки процес, котрий викликав відповідний метод, а сам процес виконання методу не потребує міжпроцесного обміну даними. Кожний *процес може робити операції над будь-якими групами*, у тому числі над такими, *до яких він навіть не входить*. При операціях над групами може виникнути порожня група – `GROUP_EMPTY`.

Створення довільної групи `group`, яка відповідає якому-небудь комунікатору виконується за допомогою методу класу `COMM` з оголошенням:

```
public Group Group() throws MPIException
```

і як завжди метод є аналогом процедури `MPI_COMM_GROUP`.

Як відомо, після запуску паралельного застосування всі його процеси працюють у єдиному нетривіальному комунікаторі `COMM_WORLD`, тому є повністю логічним починати створення групи процесів з тієї групи, яка відповідає цьому комунікатору. Це можна зробити за допомогою наступного виклику:

```
Group group = MPI.COMM_WORLD.Group();
```

Якщо якась група процесів вже існує, то з неї можна створити нову групу (навіть не одну), вказавши ранги процесів, які повинні ввійти до нової групи. З метою подібного створення групи, до класу `Group` включений метод-аналог процедури `MPI_GROUP_INCL`, який має наступне оголошення:

```
public Group Incl(int[] ranks) throws MPIException
```

За допомогою єдиного параметра методу – масиву цілих чисел `ranks` до методу *передаються ранги процесів вихідної групи, які повинні бути включені у нову групу*. Створена таким чином група містить `n` процесів з рангами від `ranks[0]` до `ranks[n - 1]`. Процес, що має ранг `i` у новій групі є процесом з рангом `ranks[i]` у вихідній групі. *Всі* `n` *елементів масиву* `ranks` *повинні бути коректними і доступними рангами процесів*, *крім того не повинно бути повторюваних рангів*, у противному випадку вкидається виняток. Якщо при виклику методу у якості параметра йому передати масив `ranks` нульової довжини, тобто оголосити `int[] ranks = new int[0]`, то буде створена порожня група `GROUP_EMPTY` (перевірка – `if(group.equals(MPI.GROUP_EMPTY))`) Метод може використовуватись з метою виконання переупорядкування процесів у новій групі і встановлення порядку відмінного від порядку в старій.

Нову групу зі старої можна створити і за допомогою подібного методу, який є аналогом процедури `MPI_GROUP_EXCL` і має оголошення:

```
public Group Excl(int[] ranks) throws MPIException
```

Метод також *створює нову групу*, але, на відміну від попереднього, *тільки з тих процесів ранги, яких не входять до множини чисел в масиві* `ranks`, причому *порядок процесів*, що увійшли до нової групи, *відповідає порядку процесів у старій групі*. *Якщо* при виклику методу *вказати нульову довжину масиву* `ranks`, то *буде створена група повністю ідентична старій групі*.

В наступному фрагменті коду програми на основі процесів деякої групи `group` створюється дві інші групи процесів `group1` і `group2`, в яких множини рангів їхніх процесів не пересікаються. У кожну знову створювану групу увійде приблизно половина процесів вихідної групи `group` (при непарному числі процесів у групу `group2` увійде на один процес більше). Порядок нумерації процесів у створених групах збережеться.

```
size1 = size / 2;
for(i = 0; i < size1; i++) ranks[i] = i;
group1 = group.Incl(ranks);
```

```
group2 = group.Excl(ranks);
```

С точки зору структури груп них можна розглядати як абстрактні множини цілих чисел, які є рангами тих процесів, що входять у групу. В зв'язку з цим, в стандарті MPI передбачені такі операції створення нових груп на базі вже існуючих, які є характерними для операцій над множинами. Стандарт визначає три операції над групами процесів, як над множинами: `MPI_GROUP_INTERSECTION`, `MPI_GROUP_UNION` і `MPI_GROUP_DIFFERENCE`. Через особливості нумерації процесів, *ні об'єднання*, *ні перетинання груп*, *як множин*, *не є комутативними*, *але завжди є асоціативними*. Пакет MPJ в класі `Group` оголошує три статичні методу-члена класу, які є повними аналогами вказаних процедур. Методи мають наступні оголошення:

```
public static Group Intersection(Group group1, Group group2)
                                                throws MPIException
public static Group Union(Group group1, Group group2)
                                                throws MPIException
public static Group Difference(Group group1, Group group2)
                                                throws MPIException
```

Перший з методів – `Intersection` повертає посилання на нову створену групу, яка являє собою перетинання груп `group1` й `group2`. Отримана *група містить всі процеси групи* `group1`, *що входять також і у групу* `group2`, *процеси у новій групі упорядковані так*, *як у першій групі*.

Другий метод – `Union` *створює нову групу шляхом об'єднання груп* `group1` *і* `group2`. Група, яка отримується, містить всі процеси з обох груп. Процеси *упорядковуються таким чином*, що *спочатку розташовуються процеси групи* `group1` *у колишньому порядку*, *а за ними розташовуються*, *також у колишньому порядку*, *процеси групи* `group2`, що не ввійшли у групу `group1`.

І третій метод – `Difference` створює нову групу, яка є різницею процесів з груп `group1` і `group2`. Отримана *група містить всі елементи групи* `group1`, *які не входять у групу* `group2` *й упорядковані*, *як у першій групі*.

Сказане можна пояснити наступним прикладом. Якщо припустити, що у деяку групу gr1 входять процеси з рангами 0, 1, 2, 4, 5, а в іншу групу gr2 – процеси 0, 2, 3 (нумерація процесів задана в вихідній групі, яка відповідає комунікатору COMM_WORLD). Тоді після виконання наступного фрагменту коду

```
Group newgr1 = Group.Intersection(gr1, gr2);
Group newgr2 = Group.Union(gr1, gr2);
Group newgr3 = Group.Difference(gr1, gr2);
```

у групу newgr1 увійдуть процеси 0, 2, у групу newgr2 – процеси 0, 1, 2, 4, 5, 3 і, нарешті, у групу newgr3 увійдуть процеси 1, 4, 5.

Порядок нумерації процесів в отриманих групах відповідає порядку їхнього перерахування.

За допомогою методу-члена класу Group (повний аналог процедури MPI_GROUP_SIZE) можна визначити кількість size процесів у групі group. Метод оголошений так:

```
public int Size() throws MPIException
```

Відповідно номери процесів rank у деякій групі group можна отримати за допомогою аналога процедури MPI_GROUP_RANK – методу класу Group з оголошенням:

```
public int Rank() throws MPIException
```

Якщо при виклику цього методу процес, який зробив його, не входить у групу group, то змінної, що повертається методом, привласнюється значення статичного поля класу MPI з оголошенням public static int UNDEFINED.

Клас Group, підтримуючи стандарт MPI, оголошує метод-аналог процедури MPI_GROUP_COMPARE, який призначений для порівняння двох груп

```
public static int Compare(Group group1, Group group2)
                                        throws MPIException
```

Якщо групи group1 і group2 повністю збігаються, тобто в групах містяться процеси з однаковими рангами й у тому ж самому порядку, то метод повертає значення MPI.IDENT. Якщо групи не відрізняються рангами процесів, а відмінність пояснюється тільки їхньою упорядкованістю, то повертається значення MPI.SIMILAR. І, якщо групи є повністю різними, то повертається значення MPI.UNEQUAL.

Ще один метод класу Group дає можливість зіставити ранги процесів однієї групи у відповідності з рангами процесів другої групи. Це метод-аналог процедури MPI_GROUP_TRANSLATE_RANKS з наступним оголошенням:

```
public static int[] Translate_ranks(Group group1, int[] ranks1,
                                    Group group2)
                                            throws MPIException
```

У масиві, який повертається методом, містяться ранги процесів у групі group2, які відповідають рангам процесів у групі group1, що указані в параметрі ranks1. *Масив* ranks1 *повинен містити тільки ті ранги процесів, які доступні в групі* group1.

Видалення групи, вже непотрібної для подальшого виконання застосування, може виконуватись за допомогою методу з оголошенням:

```
public void free()
```

Метод є приблизним аналогом процедури MPI_GROUP_FREE тому, що застосування цього методу не потрібне з огляду на наявність збірника сміття у віртуальній Java-машині. Метод просто присвоює посиланню на об'єкт групи значення, яке дорівнює null, але перед тим, як це буде зроблено, повністю буде завершена операція, яка вже почала виконуватись над цією групою до моменту звернення до методу free.

Наступній приклад (Ex_17) ілюструє розбиття всіх процесів застосування на дві приблизно рівні групи group1 й group2, які не перетинаються. За допомогою викликів методів Translate_ranks кожен процес з однієї групи знахо-

дить процес із тим же номером в іншій групі й за допомогою виклику методу Sendrecv обмінюється з ним повідомленням через глобальний комунікатор COMM_WORLD про власні ранги в тому ж комунікаторі. При непарному числі процесів, у групі group2 може виявитися на один процес більше ніж у групі group1, тоді останній процес із цієї групи не буде обмінюватися даними з жодним процесом з групи group1. Наприкінці програми посиланням на групи, які вже не потрібні далі, за допомогою викликів методу free привласнюється значення null, що не є обов'язковим.

```
import mpi.*;
import java.util.*;
class Ex_17 {
  private int size, size1, grRanks[], r1[], r2[];
  private Group group;
  private static int rank, root = 0, arrLength;
  private static String[] arg;
  private static Ex_17 ex;
  public Ex_17(String[] args) {
    arg = MPI.Init(args);
    size = MPI.COMM_WORLD.Size();
    rank = MPI.COMM_WORLD.Rank();
    group = MPI.COMM_WORLD.Group();
    size1 = size / 2;
    grRanks = new int[size1];
    r1 = new int[size1];
    r2 = new int[size1];
    arrLength = Integer.parseInt(arg[0]);
  }
  private void run() {
    int rank1, rank2, rank3;
    int[] a = new int[arrLength], b = new int[arrLength];
    Status status;
    Group group1, group2;
    for(int i = 0; i < size1; i++) grRanks[i] = i;
    group1 = group.Incl(grRanks);
    group2 = group.Excl(grRanks);
    rank1 = group1.Rank();
    rank2 = group2.Rank();
    if (rank1 != MPI.UNDEFINED) r1[rank1] = rank1;
    if ((rank2 != MPI.UNDEFINED) && (rank2 < size1))
```

```
    r2[rank2] = rank2;
  r1 = Group.Translate_ranks(group2, r1, group);
  r2 = Group.Translate_ranks(group1, r2, group);
  if (rank1 == MPI.UNDEFINED) {
    if (rank2 < size1) rank3 = r2[rank - size1];
    else rank3 = MPI.UNDEFINED;
  }
  else rank3 = r1[rank];
  Arrays.fill(a, rank);
  Arrays.fill(b, MPI.UNDEFINED);
  if (rank3 != MPI.UNDEFINED)
    status = MPI.COMM_WORLD.Sendrecv(a, 0, a.length, MPI.INT,
                                     rank3, 1,
                                     b, 0, b.length, MPI.INT,
                                     rank3, 1);
  group.free(); group1.free(); group2.free();
  System.out.printf(" %nПроцесс %d%n  a[] = %s%n  b[] = %s%n",
                rank, Arrays.toString(a), Arrays.toString(b));
}
public static void main(String[] args) throws MPIException{
  boolean canDo = true;
  try { ex = new Ex_17(args); }
  catch(ArrayIndexOutOfBoundsException e) {
    canDo = false;
    if (rank == root)
      System.err.printf(" %nНе вистачає параметра для" +
                        " методу main.%nСинтаксис команди" +
                        " запуску програми:%n %n\t\t" +
                        "mpjrun.sh -np <К-ТЬ_ПРОЦ> Ex_17" +
                        " <РОЗМІР_МАСИВУ>");
  }
  catch(NumberFormatException e) {
    canDo = false;
    if (rank == root)
      System.err.printf(" %nПомилка у командному рядку.%n" +
                        "Параметр програми \"%s\" не є" +
                        " числом!", arg[0]);
  }
  finally {
    if (canDo) ex.run();
    MPI.Finalize();
  }
}
```

```
}
```

5.5.2 Операції з комунікаторами

Як вже підкреслювалося комунікатор створює окремий контекст для обміну даними між процесами деякої групи. Такий контекст забезпечує можливість незалежних обмінів даними. У загальному випадку, *кожній групі процесів може відповідати декілька комунікаторів*, але кожен комунікатор у будь-який момент часу однозначно відповідає тільки одній групі, тобто *одночасно комунікатор не може входити у різні групи*.

Як вже говорилось раніш, одразу після виклику методу `Init` в класі `MPI` ініціюються три статичних поля-члена з такими оголошеннями:

```
public static Intracomm COMM_WORLD;
public static Comm COMM_SELF;
public static Comm COMM_NULL;
```

тим самим створюються три комунікатори перший з яких – `COMM_WORLD` поєднує всі процеси застосування, другий – `COMM_SELF` містить у себе тільки процес, який викликав метод `Init` і останній – `COMM_NULL` слугує для визначення помилково створеного комунікатору.

На відміну від створення групи, *створення комунікатора є колективною операцією* і потребує операції міжпроцесного обміну, тому такі методи повинні викликатися всіма процесами деякого існуючого комунікатора.

В стандарті MPI передбачена процедура створення нового комунікатора з тією ж групою процесів з такими ж атрибутами, що й у вихідному комунікаторі – це процедура `MPI_COMM_DUP`. *В жодному класі пакета* MPJ *не міститься методу з подібним ідентифікатором*, *але метод* `clone` *класу* `Object`, *який перевизначається у класі* `Comm`, *насправді є повним аналогом цієї процедурі і дозволяє отримати новий комунікатор-копію*, *але такий*, *що має свій власний контекст для обміну даними*.

Новий комунікатор на базі існуючої групи процесів group, яка є підмножиною групи, що пов'язана з деяким іншим комунікатором, можна створити за допомогою аналога процедури MPI_COMM_CREATE – методу класу Intracomm з наступним оголошенням:

```
public Intracomm Create(Group group) throws MPIException
```

Оскільки створення нового комунікатора є колективною операцією, то виклик цього методу повинен зустрітися у всіх процесах вихідного комунікатора. Процесам, які не належать до групи group, у параметрі, який вертається методом, буде привласнено значення статичного поля класу MPI, яке оголошено як public static COMM_NULL.

У наступному фрагменті програмного коду створюється дві нових групи, одна з яких містить першу половину процесів, а інша − другу половину. При непарному числі процесів у другу групу ввійде на один процес більше. Кожна група створюється тільки на тих процесах, які до неї входять. Для кожної нової групи створюється відповідний їй комунікатор, і виклик методу Allreduce виконується окремо для процесів, що входять у різні групи (комунікатори).

```
group = MPI.COMM_WORLD.Group();
groupSize = size / 2;
int[] ranks = new int[groupSize];
for(i = 0; i < groupSize; i++) ranks[i] = i;
if(rank < groupSize) new_group = group.Incl(ranks);
else new_group = group.Excl(ranks);
new_comm = Create(new_group);
new_comm.Allreduce(sbuf, 0, rbuf, 0, 1, MPI.INT, MPI.SUM);
new_rank = new_group.Rank();
System.out.printf(" %nrank = %d%nnewrank = %d%nrbuf[0] = %d",
                  rank, new_rank, rbuf[0]);
```

З існуючого комунікатора можна створити декілька нових застосувавши аналог процедури MPI_COMM_SPLIT – метод класу Intracomm

```
public Intracomm Split(int color, int key) throws MPIException
```

Параметр color, який передається у метод слугує для визначення процесів які попадуть в той чи інший новий комунікатор. Процеси, що викликали метод з однаковим значенням цього параметра (операція колективна і повинна виконуватися всіма процесами), утворюють один комунікатор. Параметр key визначає призначення рангів процесам у новому комунікаторі. Процеси, які викликали метод з більшим значенням параметра key одержать більший ранг у новій групі, при однаковому значенні параметра key порядок нумерації процесів вибирається системою випадково.

Якщо який-небудь з процесів не повинний ввійти в жоден з нових комунікаторів, то у якості параметра color слід указувати константу визначену як статичне поле класу MPI – public static int UNDEFINED. Після повернення з методу таки процеси одержать значення MPI.COMM_NULL.

Виконання в програмі наступного оператора

```
new_comm = MPI.COMM_WORLD.Split(rank % 3, rank);
```

призведе до створення на базі комунікатора COMM_WORLD трьох нових. До першого комуникатора ввійдуть процеси для яких параметр color дорівнює 0, тобто процеси з номерами 0, 3, 6, …. У другій комунікатор попадуть процеси з color == 1 (відповідні ранги – 1, 4, 7, …). І, врешті, у третій потраплять процеси з рангами 2, 5, 8 і т.д (color == 2). Передача у метод в якості параметра key змінної rank гарантує, що порядок нумерації процесів у створюваних групах буде відповідати порядку нумерації у вихідній групі, тобто, порядку вказаному вище.

На відміну від інших випадків, для видалення комунікаторів в класі Comm передбаченій метод-аналог процедури MPI_COMM_FREE, який оголошується так:

```
public void Free() throws MPIException
```

Після виконання методу, комунікатору для якого він викликаний привласнюється значення помилкового комунікатора MPI.COMM_NULL. Якщо, на момент

виклику методу, із цим комунікатором вже виконується якась комунікаційна операція, то вона обов'язково повинна бути завершеною раніш ніж комунікатор буде видалений.

Наступний приклад (Ex_18) демонструє створення з комунікатора COMM_WORLD нового комунікатора comm_revs, у який входять всі процеси застосування пронумеровані у зворотному порядку. Коли комунікатор стає непотрібним, він видаляється за допомогою виклику методу Free. Наведений приклад демонструє можливість використання методу Split для зміни нумерації процесів.

```
import mpi.*;
class Ex_18 {
  private int rank, size;
  private Intracomm comm_revs;
  public Ex_18(String[] args) {
    MPI.Init(args);
    size = MPI.COMM_WORLD.Size();
    rank = MPI.COMM_WORLD.Rank();
  }
  private void doSplit() {
    comm_revs = MPI.COMM_WORLD.Split(1, size - rank);
    int rank1 = comm_revs.Rank();
    String format = " %nУ комунікаторі \"COMM_WORLD\" ранг =" +
                    " %d, а у \"comm_revs\" ранг = %d.";
    System.out.printf(format, rank, rank1);
    comm_revs.Free();
  }
  public static void main(String[] args) throws MPIException {
    Ex_18 ex = new Ex_18(args);
    ex.doSplit();
    MPI.COMM_WORLD.Barrier();
    if (ex.rank == 0) System.out.printf("%n %n");
    MPI.Finalize();
  }
}
```

5.5.3 Контрольні запитання та завдання для самостійної роботи

− Дайте визначення групи. Які об’єкти системи створюють групу, чим характеризується група?

− Назвіть групи процесів, які з самого початку створюються системою при запуску застосування? Дайте їхню розгорнуту характеристику.

− Що можна сказати про ранги процесів у групах і відповідному комунікаторі?

− На основі яких об’єктів системи створюються нові групи, операції по створенню груп є колективними або локальними операціями?

− До якого класу належать методи по створенню груп? Назвіть методи за допомогою яких можуть створюватись нові групи.

− Дайте докладний опис методів `Incl` та `Excl`. Як створити порожню групу і як перевірити, що деяка створена раніш група є пустою? Як створити повну копію деякої групи? Чи може тільки один процес у деякій групі викликати метод `Incl`?

− Як Ви думаєте може або ні в якійсь групі не бути процесу з рангом 0, а може в яку-небудь групу не ввійти процес, що у комунікаторі `COMM_WORLD` має ранг 0?

− Які операції можна виконувати над групами як над абстрактними множинами цілих чисел, яке обмеження накладається на ці операції?

− Подумайте і обґрунтуйте власну відповідь на запитання: можуть або ні групи процесів, як множини, мати непусте перетинання, котре повністю не співпадає з жодною з груп? І якщо відповідь «Так», то поясніть як це зробити.

− За допомогою яких методів можна довідатися про кількість процесів у групі і про їхні ранги? Членами якого класу є ці методи? Яке значення повертається процесу, що намагається дізнатися про свій ранг у групі до якої він не входить?

− Як порівняти склад двох груп, які значення і в яких випадках можуть повертатися з методу порівняння?

– Розкажіть для чого призначений і як використовується метод `Translate_ranks`.

– Що таке комунікатор, для чого він призначений і в чому полягає відмінність комунікатора от групи процесів? Назвіть які комунікатори створюються одразу при запуску додатка.

– Як Ви думаєте, можуть або ні обмінюватися даними процеси, що належать різним комунікаторам?

– Розкажіть про процес створення комунікатора, є ця дія локальною або колективною операцією. Може або ні тільки один процес у деякому комунікаторі викликати метод `Create`? Згадайте як створити повну копію будь якого комунікатора.

– Які ще методи існують, крім створення копії, для створення нового комунікатора? Для чого потрібен параметр `color` в одному з методів?

– Чи можна за допомогою методу `Split` створити рівно один новий комунікатор, а стільки нових комунікаторів, скільки процесів входить у базовий комунікатор?

– Як можна знищити вже непотрібний на даний момент часу комунікатор?

– Напишіть програму, яка розбиває всі процеси застосування на три довільних групи і в глобальному комунікаторі `COMM_WORLD` друкує ранги тих процесів, що потрапили до двох перших групи, але не потрапили до третьої.

– Реалізуйте програмну розбивку процесів на дві групи, в одній з яких здійснюється обмін даними по кільцю, а в іншій – комунікації за схемою `master-slave`.

5.6 Створення і робота з віртуальними топологіями процесів

В стандарті MPI під поняттям ***топологія*** розуміють механізм зіставлення процесам деякого комунікатора альтернативної схеми адресації. В MPI топології є в повному розумінні ***віртуальними топологіями***, тобто вони не яким чином пов’язані з фізичною топологією комунікаційної мережі. За звичаєм топологія

використовується програмістами для більш зручного позначення процесів і, таким чином, наближення паралельної програми до структури математичного алгоритму. Крім того, за рахунок зміни порядку нумерації процесів усередині комунікатора, топологія може використовуватись системою для оптимізації розподілу процесів по фізичних процесорах існуючого паралельного комп'ютера.

В MPI передбачені два типи топології:

– ***декартова топологія*** (прямокутні ґрати довільної розмірності);

– ***топологія графа*** (дерева, ациклічні графи).

Методи для створення і керування комунікаторами з тією або іншою топологією в пакеті MPJ Express зібрані у класах `Cartcomm` і `Graphcomm`, а, також, деякі з методів успадковуються від суперкласів `Comm` і `Intracomm` (див. дерево класів пакету рис. 4.3)

За допомогою аналога процедури `MPI_TOPO_TEST` – методу класу `Comm` і перевизначеного в класах `Cartcomm` і `Graphcomm`

```
public int Topo_test() throws MPIException
```

можна визначити тип топології, яка пов'язана з деяким комунікатором.

Для комунікатора, який одержав повідомлення, що відповідає даному методу, вертається значення `MPI.GRAPH`, якщо комунікатор має топологію графа і `MPI.CART`, якщо з комунікатором пов'язана декартова топологія. Повернення значення `MPI.UNDEFINED` означає, що комунікатору не привласнений будь-який тип топології.

5.6.1 Створення декартової топології і робота з нею

Створення з процесів деякого вихідного комунікатора нового комунікатора `comm_cart`, якому властива декартова топологія, виконується за допомогою методу-члену класу `Intracomm`, який є аналогом процедури `MPI_CART_CREATE`

```
public Cartcomm Create_cart(int[] dims, boolean[] periods,
                            boolean reorder)
```

```
throws MPIException
```

Довжина масиву `dims` – `dims.length`, тобто кількість елементів у цьому масиві, визначає розмірність одержуваних декартових ґрат, а значення безпосередньо елементів цього масиву – `dims[i]` задають кількість процесів у відповідному вимірі `i`. Булеві елементи у масиві `periods`, який має ту ж саму довжину, що і масив `dims`, при значенні елементу, яке дорівнює `true`, визначають те, що ґрати є періодичними уздовж відповідного `i`-го виміру, або ні при значенні елементу `false`. *Періодичність виміру практично означає його замкнутість у кільце*. Булевий параметр `reorder`, якщо його значення дорівнює `true`, визначає, що системі, з метою оптимізації розподілу процесів по фізичних процесорах наявного паралельного комп'ютера, дозволено міняти порядок нумерації процесів в такому картезіанському комунікаторі.

Метод є колективним, тому повинний викликатись всіма процесами комунікатора. Якщо кількість процесів у новому комунікаторі `comm_cart`, для якого задається топологія, менше числа процесів у вихідному комунікаторі, то деяким його процесам повертається значення `MPI.COMM_NULL`, отже вони не приймають участі в створюваній топології. В противному випадку, *коли кількість процесів у створюваній топології більше числа процесів у вихідному комунікаторі*, то виклик є помилковим і метод *вкидає відповідний виняток*.

Наступний фрагмент програмного коду демонструє випадок створення тримірної топології 4×3×2, кожен вимір якої є періодичним, крім того, системі дозволяється змінювати порядок процесів. Даний фрагмент повинен виконуватися на кількості процесів, яка повинна бути не меншою за 24.

```
Cartcomm comm_cart;
int dims[] = {4, 3, 2};
boolean periods[] = new boolean[3];
Arrays.fill(periods, true);
comm_cart = MPI.COMM_WORLD.Create_cart(dims, periods, true);
```

Допомогти у визначенні найбільш вдалого розподілу процесів за напрямками координат в картезіанському комунікаторі може метод-аналог процедури MPI_DIMS_CREATE. Цей член класу Cartcomm має наступне оголошення:

```
public static void Dims_create(int nnodes, int[] dims)
                                        throws MPIException
```

Перший параметр методу – nnodes визначає кількість доступних процесів для створення нового картезіанського комунікатора. Другий параметр – масив dims одночасно є вхідним й вихідним параметром методу. На вході в цей масив заноситься деяке попереднє розподілення процесів по координатах, на виході з методу масив містить більш вдалий розподіл.

Метод генерує виняток у разі коли масив dims містить від'ємні елементи (хоча б один), або у разі якщо не виконується наступна умова:

$$\prod_{i,\, dims[i] \neq 0} dims[i].$$

Найбільш принадною вважається така декартова топологія, у якій число процесів по різним вимірам є приблизно однаковим. Використовуючи вказаний метод програміст може управляти числом процесів у деяких вимірах ґрат у такий спосіб: якщо перед викликом яке-небудь значення масиву dims[i] дорівнює 0, на виході в цьому елементі повертається найбільш вдале розраховане у методі значення. У противному випадку значення залишається без зміни. Негативні значення елементів масиву dims є помилковими. Перед викликом методу розмір масиву dims.length повинен бути кратним добутку ненульових значень масиву dims. Вихідні значення масиву dims, які перевизначені даним методом, будуть упорядковані в порядку убування. Метод є локальним й не потребує міжпроцесної взаємодії.

У табл. 5.3 наведені чотири приклади, які пояснюють використання методу Dims_create для створення деякої тримірної топології. У першому прикладі з 6 процесів можна утворити ґрати 3×2×1, причому розміри впорядковані в порядку убування. У другому прикладі робиться спроба розподілити 7 процесів по

трьох вимірах, єдиний можливий в цьому разі варіант − ґрати 7×1×1. У третьому прикладі для другої розмірності з самого початку задане значення 3, дві розмірності, що залишаються, визначають ґрати 2×3×1. Четвертий виклик помилковий, тому що загальне число процесів – 7 не можна поділити без залишку на розмір, що заданий для другої розмірності – 3.

Таблиця 5.3 − Пояснення застосування методу `Dims_create`

`dims` до виклику	Виклик методу	`dims` після виклику
{0, 0, 0}	`Cartcomm.Dims_create(6, dims)`	{3, 2, 1}
{0, 0, 0}	`Cartcomm.Dims_create(7, dims)`	{7, 1, 1}
{0, 3, 0}	`Cartcomm.Dims_create(6, dims)`	{2, 3, 1}
{0, 3, 0}	`Cartcomm.Dims_create(7, dims)`	виняток

Нумерація координат процесів у декартовії системі починається з нуля. Причому спочатку вказується номер рядка, до якого належить процесор, а потім стовпчика. Це означає, що взаємозалежність між рангами і координатами чотирьох процесів у ґратах розмірністю 2 × 2 описується наступним чином:

coord (0, 0): rank 0

coord (0, 1): rank 1

coord (1, 0): rank 2

coord (1, 1): rank 3

Хоча вказана залежність дуже проста, але не завжди, особливо при великій кількості координат, можна швидко визначити координати процесу за його рангом, або, навпроти, за координатами процесу довідатися про його ранг.

Тому коли у програми виникає потреба за рангом процесу визначити його картезіанські координати у комунікаторі, то зробити це дозволяє метод, який є аналогом процедури `MPI_CART_COORDS` і має таке оголошення:

```
public int[] Coords(int rank) throws MPIException
```

Метод у вихідному масиві з числом елементів, яке дорівнює кількості вимірів ґрат, повертає значення координат процесу з вказаним рангом `rank`. Відлік координат за кожним виміром, як вже відмічалося, починається з нуля.

Якщо виникає протилежна задача − визначення рангу процесу в комунікаторі за його картезіанськими координатами, то це можна вирішити за допомогою методу-аналога процедури `MPI_CART_RANK` з наступним оголошенням:

```
public int Rank(int[] coords) throws MPIException
```

Для ґрат, які є періодичними, координати поза припустимими інтервалами перераховуються у відповідні координати по кільцю, а для неперіодичних ґрат вони є помилковими і системою вкидається відповідний виняток.

В разі необхідності, поділ комунікатора, з яким зв'язана деяка декартова топологія, на підгрупи, що відповідають картезіанським підґратам меншої розмірності, можна зробити методом, який є аналогом процедури `MPI_CART_SUB` і оголошується як

```
public Cartcomm Sub (boolean[] remain_dims) throws MPIException
```

До методу передається масив `remain_dims` булевих значень, розмір якого відповідає розмірності вихідного комунікатора. Елементи в масиві повинні приймати значення `true` для тих вимірів, які без змін переноситься у нові комунікатори меншої розмірності.

Якщо повернутися до прикладу створення комунікатора з тримірною топологією 4×3×2, який був розглянутий вище (див. стор. 170), то можна зробити спробу поділити цей комунікатор на 3 нових комунікатора розмірністю 4×2, які містять по 8 процесів кожний. Нижче показано, як для цього застосовується метод `Sub`.

```
boolean remain_dims[] = {true, false, true};
newcomm = comm_cart.Sub(remain_dims);
```

Аналог одразу двох процедур MPI_CARTDIM_GET і MPI_CART_GET – метод з оголошенням:

```
public CartParms Get() throws MPIException
```

в полях екземпляра об'єкта класу CartParms повертає інформацію про об'єкт декартового комунікатора, який одержав повідомлення, яке відповідає цьому методу.

Клас CartParms має наступний опис:

```
public class CartParms {
  CartParms() {}
  public int[] dims;
  public boolean[] periods;
  public int[] coords;  // coords[0] відповідає осі X,
                        // coords[1] –осі Y...
}
```

Таким чином, у полі dims отриманого об'єкта повертається кількість процесів для кожного виміру, у полі periods − періодичність по кожному виміру і, нарешті, в полі coords містяться координати у декартовій топології процесу, що викликав цей метод. Розмірність картезіанської топології комунікатора може бути визнана шляхом одержання розміру масова dims (dims.length).

Для визначення рангів процесів, які повинні здійснювати обмін повідомленнями вздовж якого-небудь виміру direction на відстані disp між процесом-відправником і процесом-приймачем повідомлення у багатомірному комунікаторі слугує метод-аналог процедури MPI_CART_SHIFT з описом:

```
public ShiftParms Shift(int direction, int disp)
                                        throws MPIException
```

Метод повертає екземпляр об'єкта класу ShiftParms з описом:

```
public class ShiftParms {
```

```
    ShiftParms() {}
    public int rank_source;
    public int rank_dest;
  }
```

у полях rank_dest (приймач) і rank_source (відправник) якого містяться ранги відповідних процесів. Для періодичних вимірів здійснюється циклічне зміщення вздовж виміру, а для неперіодичних − лінійне. У випадку лінійного зсуву для деяких номерів процесів, які відправляють або отримують повідомлення, може бути отримане значення MPI.PROC_NULL, що означає вихід за межі діапазону процесів вздовж координати. У випадку циклічного зсуву останній процес у даному вимірі здійснює обміни з нульовим процесом. *Для* n-*мірних декартових ґрат значення* direction *повинно бути в межах від* 0 *до* n-1.

Одержані значення rank_source і rank_dest можна, наприклад, використати для обміну за допомогою методу Sendrecv, що й робиться у черговому фрагменті коду програми, який демонструє такий алгоритм роботи. По-перше, створюються деякі періодичні по обох вимірах двомірні картезіанські ґрати (отримується такий собі тор), визначаються ранг і координати кожного процесу в отриманих ґратах. Потім за допомогою методу Shift обчислюються координати процесів, з якими передбачається виконання обміну даними за алгоритмом здійснення циклічного зсуву з кроком 2 у першому вимірі. Наприкінці фрагмента отримані значення рангів процесів використовуються для безпосереднього обміну даними за допомогою методу Sendrecv_replace.

```
  boolean periods[] = {true, true};
  int[] coords;
  Cartcomm commCart = MPI.COMM_WORLD.Create_cart(dims, periods,
                                                 true);
  rank = commCart.Rank();
  coords = commCart.Coords(rank);
  int shift = 2;
  ShiftParms parms = commCart.Shift(0, shift);
  Status status = commCart.Sendrecv_replace(a, 0, 1, MPI.DOUBLE,
                                            parms.rank_dest, 0,
                                            parms.rank_source, 0);
```

Прикладом створення картезіанської топології процесів і використання подібного зсуву даних від одного процесу до іншого із застосуванням методу `Shift` (точніше кажучи `MPI_CART_SHIFT`) може слугувати паралельний алгоритм вирішення задачі Пуассона, приведений авторами [4], а декілька інша версія – [5].

5.6.2 Створення і робота з віртуальною топологією графа

На практиці існує велика кількість паралельних обчислювальних алгоритмів, які потребують створення нового комунікатора так, щоб зв'язки між його процесами відповідали топології графа. Створення такої групи процесів яка є підмножиною групи процесів деякого іншого комунікатора, виконується аналогом процедури `MPI_GRAPH_CREATE` – методом класу `Intracomm`, який має наступне оголошення:

```
public Graphcomm Create_graph(int[] index, int[] edges,
                              boolean reorder)
                                          throws MPIException
```

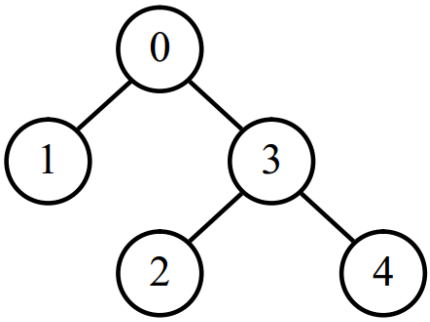
Параметр-масив `index`, який передається до цього методу, у кожному `i`-у елементі містить сумарну кількість сусідів всіх вершин до `i`-ї вершини включно. Розмір цього масиву – `index.length` відповідає кількості вершин створюваного графа (`nnodes`). Масив ребер графа – `edges` містить перелік номерів процесів-сусідів упорядкований для кожної вершини. Розмір масиву `edges` повинен дорівнювати значенню, яке зберігається у остатньому елементі масиву `index`, тобто у елементі `index[nnodes - 1]`. Якщо значення булевого параметра `reorder` дорівнює `true`, то це означає, що системі дозволено міняти порядок нумерації процесів в новому комунікаторі з метою їхнього зручнішого упорядкування.

Метод є колективним, і повинен викликатись всіма процесами вихідного комунікатора. Якщо `nnodes` менше числа процесів вихідного комунікатора, то деяким процесам повернеться значення `MPI.COMM_NULL` і вони не будуть при-

ймати участи в створюваній топології, а в разі коли `nnodes` більше кількості процесів вихідного комунікатора буде вкинутий відповідний виняток часу виконання.

Пояснення приведеного опису застосування методу `Create_graph` може бути зроблено за допомогою наступного прикладу.

Якщо припустити, що створюється комунікатор, який відповідає графу зображеному на рис. 5.12 а), то такий граф формально можна описати шляхом завдання всіх сусідів кожної вершини рис. 5.12 б).



а)

Процес	Сусіди
0	1, 3
1	0
2	3
3	0, 2, 4
4	3

б)

Рисунок 5.12 – Можливе представлення графа п’ятьох процесів
а) – зображення графа, б) – його формальний опис

А безпосередньо створення такого графа на базі вже існуючого глобального комунікатора `COMM_WORLD` може бути здійснено за допомогою нижче наведеного фрагмента коду.

```
int[] index = {2, 3, 4, 7, 8};
int[] edges = {1, 3, 0, 3, 0, 2, 4, 3};
Graphcomm comm_graph = MPI.COMM_WORLD.Create_graph(index, edges,
                                                    true);
```

На рис. 5.13 наведена схема згідно з якою виконуються розрахунки елементів перших двох параметрів, які передаються до методу `Create_graph` – масивів `index[]` та `edges[]`.

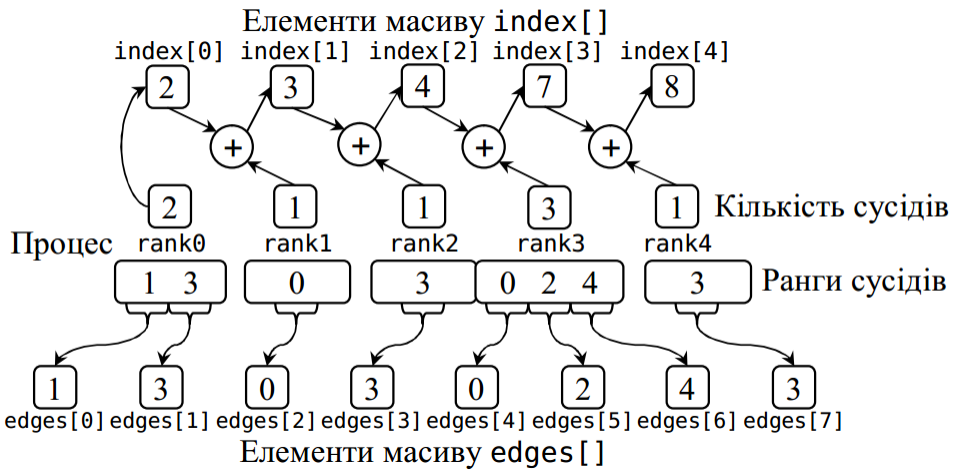


Рисунок 5.13 – Схема розрахунків елементів масивів `index` і `edges` для передачі у метод `Create_graph`

Наведений код буде коректним при виконанні у комунікаторі, який містить не менш ніж п’ять процесів.

Метод `Neighbours` – член класу `Graphcomm` має оголошення:

```
public int[] Neighbors(int rank) throws MPIException
```

і замінює собою одразу дві стандартні процедури `MPI_GRAPH_NEIGHBORS` і `MPI_GRAPH_NEIGHBORS_COUNT`.

Власно кажучи, метод є повним аналогом першої процедури тому, що він, як і процедура `MPI_GRAPH_NEIGHBORS`, *у деякому масиві* `neighbors` *повертає визначені ранги безпосередніх сусідів процесу у комунікаторі з топологією графа*, ранг `rank` цього процесу передається до методу як вхідний параметр.

Включення у клас додаткового методу-члену, дія якого подібна до дії процедури `MPI_GRAPH_NEIGHBORS_COUNT`, є надлишковим тому, що кількість безпосередніх сусідів процесу з рангом `rank` у комунікаторі з топологією графа, до якого він входить, можна з легкістю довідатися одержавши розмір масиву `neighbors.length`.

Інформація про топологію графа, яка пов’язана з деяким комунікатором, одержується методом `Get` класу `Graphcomm` з оголошенням:

```
public GraphParms Get() throws MPIException
```

Метод знов-таки, як і попередній, є подвійним аналогом, але вже для процедур MPI_GRAPH_GET і MPI_GRAPHDIMS_GET, точніше кажучи, повним аналогом першої з них. В полях екземпляра об'єкта класу GraphParms з описом:

```
public class GraphParms
  {
    public int[] edges;
    public int[] index;
  }
```

цей метод *повертає опис топології графа у тому вигляді, як вона задається при створенні* за допомогою методу Create_graph.

Визначення ж кількості вершин і числа ребер графа, замість аналога надлишкової в цьому випадку процедури MPI_GRAPHDIMS_GET, можуть бути одержані з обчислення розмірів масивів index.length і edges.length, відповідно.

У наступній програмі (Ex_19) для комунікатора comm_graph створюється така топологія графа, яка дозволяє спілкування між процесами за комунікаційною схемою master-slave. Всі процеси в рамках даної топології можуть спілкуватися тільки з нульовим процесом. Створивши відповідну топологію викликом методу Create_graph, кожен процес за допомогою звернення до методу Neighbors визначає кількість і ранги своїх безпосередніх сусідів у межах даної топології. Після цього кожен процес вже може в середині такого новоствореного комунікатора обмінюватися даними зі своїми безпосередніми сусідами. Наприклад, у прикладі програми спілкування ведеться із застосуванням звернення до методу одночасного прийому/передачі повідомлень Sendrecv.

```
import mpi.*;
import java.util.*;
class Ex_19 {
  private int size, dataLength,
              sendBuf[], recvBuf[];
  private static int rank;
```

```
private static String[] arg;
private static Ex_19 ex;
public Ex_19(String[] args) {
  arg = MPI.Init(args);
  size = MPI.COMM_WORLD.Size();
  if (size < 2) throw new IllegalArgumentException();
  rank = MPI.COMM_WORLD.Rank();
  dataLength = Integer.parseInt(arg[0]);
  sendBuf = new int[dataLength];
  Arrays.fill(sendBuf, rank);
  recvBuf = new int[dataLength];
}
private void masterSlave() {
  Status status;
  int tag = 55;
  int[] index = new int[size], neighbors = new int[size],
        edges = new int[size * 2 - 1];
  for(int i = 0; i < size; i++) index[i] = size + i - 1;
  for(int i = 0; i < size - 1; i++) {
    edges[i] = i + 1;
    edges[size + i] = 0;
  }
  Graphcomm comm_graph =
              MPI.COMM_WORLD.Create_graph(index, edges, true);
  neighbors = comm_graph.Neighbours(rank);
  for(int i = 0; i < neighbors.length; i++) {
    status = comm_graph.Sendrecv(sendBuf, 0, ex.dataLength,
                                 MPI.INT, neighbors[i], tag,
                                 recvBuf, 0, ex.dataLength,
                                 MPI.INT, neighbors[i], tag);
    System.out.printf(" %nПроцес %d зв'язався з процесом" +
                      " %d,%n\tнадіслав йому %s,%n\t" +
                      "а одержав %s", rank, status.source,
                      Arrays.toString(sendBuf),
                      Arrays.toString(recvBuf));
  }
}
public static void main(String[] args) throws MPIException {
  boolean canDo = true;
  try { ex = new Ex_19(args); }
  catch (ArrayIndexOutOfBoundsException e) {
    canDo = false;
    if (ex.rank == 0)
```

```
        System.err.printf(" %nНе вистачає параметра для" +
                          " методу main.%nСинтаксис команди" +
                          " запуску програми:%n %n\t\t" +
                          "mpjrun.sh -np <К-ТЬ_ПРОЦ> Ex_19" +
                          " <РОЗМІР_БУФЕРУ>%n %n");
      }
      catch (NumberFormatException e) {
        canDo = false;
        if (ex.rank == 0)
          System.err.printf(" %nПомилка у командному рядку.%n" +
                          "Параметр програми \"%s\" не є" +
                          " числом!%n %n", arg[0]);
      }
      catch(IllegalArgumentException e) {
        canDo = false;
        if (rank == 0)
          System.err.printf(" %nНемає підлеглих процесів.%n" +
                          "Кількість процесів повинна бути" +
                          " більшою за 1%n %n");
      }
      finally {
        if (canDo) ex.masterSlave();
        MPI.Finalize();
        if (ex.rank == 0) System.out.printf("%n %n");
      }
    }
  }
```

5.6.3 Контрольні запитання та завдання для самостійної роботи

− Що мається на увазі у стандарті MPI під терміном «топологія», чому топології у стандарті віртуальні? З якою метою у стандарт введений такий механізм?

− Назвіть два типи топологій, які реалізовані у стандарті і в пакеті MPJ Express. Членами яких класів є методи, що призначені для створення і керування топологіями процесів?

− Як можна визначити тип топології, що пов’язана з деяким комунікатором, які значення вертаються відповідним методом?

– Докладно опишіть метод створення картезіанської топології `Create_cart`. Що означає періодичність того чи іншого виміру? Можна або ні викликати метод тільки на половині процесів комунікатора?

– Як визначити найбільш вдалий розподіл процесів за напрямками координат в картезіанському комунікаторі? У яких випадках при користуванні відповідним методом вкидається виняток типа `MPIException`?

– Завжди або ні є необхідним використовувати метод `Dims_create` перед тім як звертатися до методу створення декартової топології?

– Який порядок нумерації координат процесів у декартовії системі передбачається у стандарті MPI?

– Як за рангом процесу визначити його картезіанські координати у комунікаторі? А як виконати зворотну задачу – за координатою у ґратах визнати ранг процесу?

– Який параметр, ранг або координата процесу, використовується в декартовій топології в методах обміну даними? Обґрунтуйте свою відповідь.

– Опишіть метод за допомогою якого можна виконати поділ картезіанського комунікатора на декілька підгруп меншої розмірності.

– Чим метод `Get` бібліотеки класів MPJ Express відрізняється від методів, що передбачаються стандартом MPI, які параметри ґрат він повертає?

– В чому полягає призначення метода `Shift`, які пересилання даних ним здійснюються? Докладно розкажіть про метод.

– Яким методом класу `Intracomm` створюється група процесів так, що зв'язки між ними відповідають топології графа? Докладно опишіть цей метод.

– Для групи процесів з довільною топологією графа проілюструйте процес розрахунків елементів у масивах `index[]` та `edges[]` для передачі них у якості фактичних параметрів у метод `Create_graph`.

– Подумайте як створити топологію графа, у якій кожен процес пов'язаний з кожним (повнозв'язана топологія). Спробуйте написати відповідну програму.

– Яких два методи стандарту MPI замінює метод `Neighbors` класу `Graphcomm`? Для чого і як він застосовується?

– Яким чином у бібліотеці класів пакету MPJ Express можна одержати інформацію про топологію графа і яка це інформація? Які методи передбачені для цього у стандарті MPI?

– Напишіть вихідний код, в якому виконується розбивка процесів на дві групи так, що в першій процеси організовані у одномірну декартову топологію і здійснюють обмін по кільцю зсувом, а в іншій, яка реалізує топологію графа, комунікації за схемою `master-slave`.

– Спробуйте створити паралельну програму для одержання добутку матриць, використовуючи для цього двомірну декартову топологію процесів.

5.7 Засоби пересилання різнотипних даних

В стандарті MPI *під повідомленням розуміється масив однотипних даних, розташованих у послідовних комірках пам'яті*. Але в програмах доволі часто буває необхідним пересилати більш складні структури даних, які повинні складаються з різнотипних елементів. Наприклад, одному з процесів треба направити іншому повідомлення, яке містить ціле число, а після нього низку дійсних чисел у відповідній кількості. Прикладом відправки повідомлення, коли данні розташовуються не в послідовних комірках пам'яті, може слугувати випадок пересилання підматриці, яка є частиною іншої матриці. У цих випадках можна звернутися або до посилання даних невеличкими порціями елементів одного типа, які розташовані підряд у послідовних комірках запам'ятовуючого пристрою комп'ютера, або перед відсиланням робити копіювання даних у деякий проміжний буфер для об'єднання них у одну структуру. Обидва варіанти не досить зручні і потребують додаткових витрат як часу роботи програми, так й ресурсів оперативної пам'яті. Замість цього стандарт MPI пропонує механізм створення більш узагальненого, змішаного і не безперервного комунікаційного буфера.

Виходячи зі сказаного, у бібліотеках MPI для запобігання небажаних і непередбачених витрат часу і пам'яті пропонується два спеціальних способи для виконання пересилання різнотипних даних:

– введення похідних типів даних;
– упакування даних.

5.7.1 Похідні типи даних

Похідні типи даних створюються під час виконання програми за допомогою методів-конструкторів (не плутати з конструкторами об'єктів у мові Java) на основі базових, вже існуючих до моменту виклику конструктора, типів даних. Ці способи конструювання похідних типів даних можна застосовувати рекурсивно.

Створення похідного типа даних складається із двох етапів:

– конструювання нового типа;
– реєстрація типа.

Після реєстрації похідний тип даних, поряд з наперед визначеними базовими типами, *можна використовувати в операціях пересилання*, у тому числі й у колективних операціях. *Після завершення роботи з похідним типом даних його рекомендується анулювати*. При цьому всі зроблені на його основі нові структури даних залишаються без змін й можуть надалі використовуватись у програмі.

Узагальнений похідний тип даних являє собою інкапсульований об'єкт, який визначає дві речі:

– послідовність базових типів у буфері;
– послідовність цілих чисел (байтів), які задають зміщення елементів типа відносно початку буфера обміну.

Зміщення можуть бути, як додатними, так і від'ємними, зовсім не обов'язково, щоб всі вони були різними, навпаки вони можуть бути однаковими для різних елементів послідовності, не потрібна, також, і їхня впорядкованість.

Таким чином, послідовність елементів даних у похідному типі може відрізнятися від послідовності вихідного типа, а один елемент даних може зустрічатися у створюваному типі багаторазово. Таки пари послідовностей (або послідовності пар) називаються ***шаблоном типа*** (***TypeMap***). Послідовність тільки базових типів (без урахування зміщень) має назву ***сигнатури типа*** (***TypeSig***).

Тобто шаблон типа це наступна послідовність:

$$TypeMap = \{(type_0,\ disp_0),\ ...,\ (type_{n-1},\ disp_{n-1})\}, \tag{3.1}$$

де $type_i$ – базові типи, а $disp_i$ – відповідне зміщення типа відносно начала буферу.

Тоді послідовність:

$$TypeSig = \{type_0,\ ...,\ type_{n-1}\}$$

асоціюється з поняттям *сигнатури типа*. Цей шаблон спільно з базовою адресою `buf`, визначає комунікаційний буфер, який складається з входження `n` елементів, причому кожне `i`-те входження знаходиться за адресою `buf + disp`$_i$ і має тип `type`$_i$. Повідомлення, яке пов'язане з цім комунікаційним буфером буде складатись з `n` значень, типи яких визначаються сигнатурою типа.

Попередньо визначені базові типи, які перелічені у табл. 5.1, фактично є окремим випадком узагальненого типа даних. Тобто оголошений тип `MPI.INT` умовно може розглядатись як такий шаблон: `{(int, 0)}` – входження одного значення типа `int` з нульовим зміщенням. Аналогічно можна уявляти собі і інші базові типи.

Крім шаблону і сигнатури типа даних вводиться ще поняття ***екстент типа*** – інтервал від першого до останнього байта пам'яті, яка відводиться для створюваного типа. Якщо шаблон визначається виразом (3.1), тоді нижня границя, пам'яті, що займається типом, буде

$$lb(TypeMap) = \min_j disp_j,$$

а верхня границя визначається як

$$ub\left(TypeMap\right)=\max_{j}\left(disp_{j}+sizeof\left(type_{j}\right)\right)+\varepsilon\,,$$

нарешті, екстент буде

$$extent\left(TypeMap\right)=ub\left(TypeMap\right)-lb\left(TypeMap\right). \qquad (3.2)$$

Найчастіше $type_j$ вимагає вирівнювання на парну адресу пам'яті, тому ε – це найменша не від'ємна добавка, яка потрібна для округлення екстенту типа у більший бік при вирівнюванні даних.

В пакеті MPJ для створення і роботи з даними похідних типів призначені методи класу `Datatype`. Методи, що входять у цей клас, як і методи інших класів, є аналогами відповідних процедур стандарту MPI і слугують тим самим цілям.

Створення деякого нового типа даних `newtype`, який складається з `count` послідовно розташованих елементів базового типа даних `oldtype`, виконується статичним методом-аналогом процедури `MPI_TYPE_CONTIGUOUS`. Цей метод в класі `Datatype` оголошується наступним чином:

```
public static Datatype Contiguous(int count, Datatype oldtype)
                                          throws MPIException
```

Фактично тип даних, що буде створений, являє з себе масив даних базового типа, які об'єднані у окремий об'єкт добре пристосований для операцій з обміну даними. На рис. 5.14 (рисунок $MPJ_HOME%/doc/images/mpj-section.png з офіційного, вільно розповсюджуваного по ліцензії MIT без обмеження прав використання, набору файлів пакету MPJ Express версії 0.44) наведена структура комунікаційного буфера, який відповідає створеному цім методом типа даних.

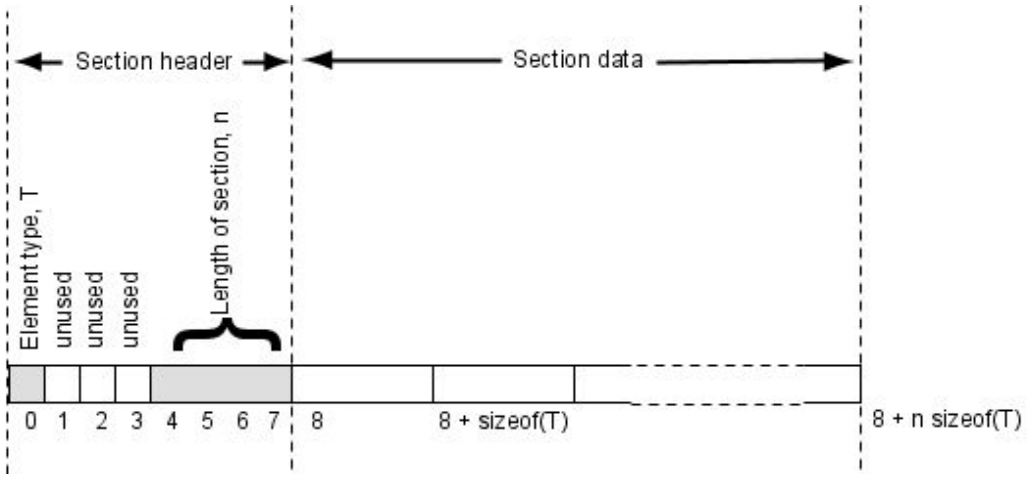


Рисунок 5.14 − Структура комунікаційного буферу для типа даних, створеного за допомогою метода `Contiguous`

Приклад звернення до методу з метою створення деякого нового типа даних `newtype`, який у подальшому після реєстрації типа може використовуватись для пересилання п’яти розташованих підряд цілих чисел виглядає так:

```
Datatype newtype = Datatype.Contiguous(5, MPI.INT);
```

Якщо дані, для яких створюється новий тип даних `newtype`, є не окремими поряд розташованими даними, а представляють з себе `count` разів повторені рівномірно розподілені блоки довжиною по `blocklength` елементів базового типа даних `oldtype`, то для створення нового типа застосовується аналог процедури `MPI_TYPE_VECTOR`. Це статичний метод класу `Datatype` з оголошенням:

```
public static Datatype Vector(int count, int blocklength,
                              int stride, Datatype oldtype)
                                          throws MPIException
```

Параметр цілого типа `stride`, який передається у метод, слугує для визначення зміщення початку наступного блоку вихідних даних відносно початку попереднього блоку. Цей *параметр визначається у кількості елементів базового типа даних*.

Нижче наведений приклад фрагменту коду для створення нового типа даних newtype. Якщо уявити, що шаблон типа, згідно з яким створюється новий тип, складається з таких елементів:

– тип даних;

– зміщення елементів даних від початку буфера пересилання,

и має наступну структуру:

```
{(MPI.FLOAT, 0), (MPI.FLOAT, 1), (MPI.FLOAT, 2), (MPI.FLOAT, 5),
 (MPI.FLOAT, 6), (MPI.FLOAT, 7)}.
```

Тобто містить шість блоків тип/зміщення, які об'єднані у дві окремих групи по три блоки кожна. Після реєстрації новостворений тип (newtype) вже може використовуватися для пересилання такої розрізненої структури в якості одного неподільного блоку з шістьох елементів даних.

```
int count = 2;
int blocklen = 3;
int stride = 5;
Datatype newtype = Datatype.Vector(count, blocklen, stride,
                                   MPI.FLOAT);
```

Ідентичний вектор, за винятком, що крок виражається безпосередньо в термінах індексу буфера, а не в одиницях старого типа. Метод Hvector – майже повністю ідентичний методу Vector, за винятком того, що параметр stride виражається безпосередньо в індексах усередині буфера, а не у кількості елементів старого типа. Оголошення методу виглядає так:

```
public static Datatype Hvector(int count, int blocklength,
                               int stride, Datatype oldtype)
                                           throws MPIException
```

Метод є аналогом процедури MPI_TYPE_HVECTOR, але в пакеті MPJ *параметр методу* stride *не вимірюється у байтах на відміну від його аналогів* в інших мовах програмування.

Якщо в програмі виникає потреба в створенні нового типа даних, який складається з count блоків не рівної довжини, як це має місто у семантиці двох попередніх методів, то слід застосовувати метод, який відповідає процедурі на ім'я MPI_TYPE_INDEXED і має наступне оголошення:

```
public static Datatype Indexed(int[] array_of_blocklengths,
                               int[] array_of_displacements,
                               Datatype oldtype)
                                             throws MPIException
```

У елементах масиву array_of_blocklengths до методу передається кількість елементів базового типа даних у кожному з окремих вихідних блоків. Кількість таких вихідних блоків повинна дорівнювати розміру цього масиву. А в елементах масиву array_of_displacements, тієї ж самої довжини, вказується, що блок під номером j починається через array_of_displacements[j] елементів базового типа oldtype від початку буфера пересилання.

Новий тип, що отримується після виконання методу, можна вважати узагальненням векторного типа даних.

Наступний фрагмент коду надає приклад створення типа даних newtype придатного для опису нижньої трикутної матриці з елементами типа double (при цьому враховується, що в мові Java масиви зберігаються по рядках).

```
for(int i = 0; i < n; i++)
  {
    array_of_blocklengths[i] = i + 1;
    array_of_displacements[i] = n * i;
  }
Datatype newtype = Datatype.Indexed(n, array_of_blocklengths,
                                    array_of_displacements,
                                    MPI.DOUBLE);
```

І знов, як і для описаного вище методу Vector, в класі оголошений метод Hindexed, який майже повністю ідентичний методу Indexed і має наступне оголошення:

```
public static Datatype Hindexed(int[] array_of_blocklengths,
                                int[] array_of_displacements,
                                          Datatype oldtype)
                                        throws MPIException
```

Метод також є аналогом відповідної процедури `MPI_TYPE_HINDEXED` і для елементів його параметра-масиву `array_of_displacements` повністю є справедливими всі ті зауваження, що були зроблені вище для параметра `stride`.

У клас `Datatype` включений ще один метод створення нового типа даних, який дозволяє об'єднувати не тільки декілька блоків даних різної довжини і однакового типа, але й блоки даних різного базового типа. Це статичний метод класу з описом:

```
public static Datatype Struct(int[] array_of_blocklengths,
                              int[] array_of_displacements,
                              Datatype[] array_of_types)
                                        throws MPIException
```

Метод є аналогом процедури `MPI_TYPE_STRUCT`.

Кількість блоків, які вибрані для створення нового типа визначає розмір першого аргументу методу – масиву `array_of_blocklengths`. Другий й третій аргументи методу `array_of_displacements` і `array_of_types` повинні мати той самий розмір. На відміну від аналогів методу для інших мов програмування, значення у масиві `array_of_displacements` виміряються не в байтах, а у кількості елементів базового типа від початку буфера пересилання. Елементи масиву `array_of_types`, які визначають тип `i`-го блоку *обов'язково повинні мати той самий базовий тип*, який мають всі елементи відповідного блоку. Інакше кажучи, що у блок повинні збиратися тільки елементи одного типа і блок за допомогою елемента масиву `array_of_types` позначається тим самим типом.

Наступний фрагмент коду програми ілюструє створення нового типа даних `newtype`, який відповідає такому шаблону:

```
{(MPI.DOUBLE, 0), (MPI.DOUBLE, 1), (MPI.DOUBLE, 2),
 (MPI.CHAR, 24), (MPI.CHAR, 25)},
```

тобто спочатку об'єднує три числа типа `MPI.DOUBLE`, а потім додає у структуру ще два символи типа `MPI.CHAR`.

Після реєстрації новостворений тип може бути використаний для пересилання як єдиного цілого п'яти елементів даних, які являють собою структуру, що зберігає в собі тип елемента даних та кількість елементів від початку буфера пересилання:

```
int[] array_of_blocklengths = {3, 2};
Datatype[] array_of_types = {MPI.DOUBLE, MPI.CHAR};
int[] array_of_displacements = {0, 24};
Datatype newtype = Datatype.Struct(2, array_of_blocklengths,
                                   array_of_displacements,
                                   array_of_types);
```

Як вже відмічалося раніш, створення нового похідного типа даних, ще не достатньо для його використання, *тип повинен бути зареєстрованим*. Реєстрація типа виконується аналогом процедури `MPI_TYPE_COMMIT` – методом `Commit` з оголошенням:

```
public void Commit() throws MPIException
```

Метод застосовується до створеного екземпляра об'єкта класу `Datatype` параметрів не має і особливих пояснень не потребує. Після реєстрації тип можна використовувати в операціях обміну нарівні з базовими типами даних. *Базові типи даних реєстрації не підлягають*.

Після використання похідного типа даних, в разі його подальшої непотрібності, він може бути знищений за допомогою методу `public void free()` або методу `public void finalize() throws MPIException`, але у мові Java цього робити зовсім не обов'язково.

Для роботи з похідними типами даних, стандарт MPI потребує включення до бібліотек декількох функцій, які надають можливість визначення таких характеристик даних, як розмір елемента, верхній і нижній поріг елементу у типі даних та інші.

Так, наприклад, визначення загального розміру `size` – кількості елементів у буфері даних відповідного типа `datatype`, виконується за допомогою методу

```
public int Size() throws MPIException
```

Метод є аналогом процедури `MPI_TYPE_SIZE`, але на відміну від неї повертає *розмір типа не у байтах*, *а у кількості елементів*, що входять у тип.

Визначення зміщення нижньої границі елемента типа даних `Datatype` від початку буфера даних знаходиться методом

```
public int Lb() throws MPIException
```

Метод знаходить і повертає найменше значення з діапазону позицій, що відповідає розташуванню елементу відносно початку буфера. Метод є аналогом процедури `MPI_TYPE_LB`.

Метод-аналог процедури `MPI_TYPE_UB` знаходить і повертає найбільше значення з діапазону позицій, що відповідає розташуванню елементу відносно початку буфера, тобто визначає зміщення верхньої границі елемента типа даних `Datatype`. Цей метод оголошений як

```
public int Ub() throws MPIException
```

Метод з оголошенням:

```
public int Extent() throws MPIException
```

повертає екстент блока елементів `Datatype`, тобто різницю між найбільшим та найменшим значенням діапазону елементів, який займається типом у буфері.

Метод є аналогом процедури з стандарту MPI – MPI_TYPE_EXTENT.

Наступний приклад (Ex_20) демонструє використання похідного типа даних для перестановки рядків матриці у зворотному порядку. Тип даних MATR_REV, що створюється методом Vector, описує локальну частину матриці даного процесу з переставленими у зворотному порядку рядками. Після реєстрації цей тип даних може використовуватись при пересиланні повідомлень. Програма працює правильно, якщо розмір матриці N є кратним числу процесів застосування.

```
import mpi.*;
class Ex_20 {
  public static void main(String[] args) throws MPIException {
    Ex_19 ex = new Ex_19();
    int rank, size, nl;
    String[] st = MPI.Init(args);
    int matrSize = Integer.parseInt(st[0]);
    double[][] a = new double[matrSize][matrSize],
               b = new double[matrSize][matrSize];
    rank = MPI.COMM_WORLD.Rank();
    size = MPI.COMM_WORLD.Size();
    nl = (matrSize - 1) / size + 1;
    ex.work(a, b, matrSize, nl, size, rank);
    MPI.Finalize();
  }

  void work(double[][] a, double[][] b, int n, int nl,
            int size, int rank) {
    int i, j, ii;
    Status status;
    for(i = 0; i < nl; i++)      for(j = 0; j < n; j++) {
        b[i][j] = 0.0;
        ii = i + 1 + rank * nl;
        a[i][j] = 100 * ii + j + 1; }
    Datatype MATR_REV = Datatype.Vector(nl, n, -n, MPI.DOUBLE);
    MATR_REV.Commit();
    status = MPI.COMM_WORLD.Sendrecv(a, 0, 1, MATR_REV,
                                     size - rank - 1, 1,
                                     b, 0, nl * n, MATR_REV,
                                     size - rank - 1, 1);
    for(i = 0; i < nl; i++)
```

```
        for(j = 0; j < n; j++)
          System.out.println("Процесс " + rank + ": " +
                             (i + rank * nl) + "\t" +
                             j + "\t"  + a[i][j] + "\t" +
                             b[i][j]);
    }
  }
```

5.7.2 Упакування даних

Не завжди зручно створювати новий тип даних для забезпечення пересилання повідомлень у програмі. Прикладом може слугувати випадок, коли пересиланню підлягають дані одного типа, але такі, що розташовані не в послідовних комірках пам'яті. Тому стандарт MPI передбачає альтернативний спосіб щодо створення похідних типів для підготовки даних до пересилання. Різнорідні або розташовані не в послідовних комірках пам'яті дані за допомогою спеціального методу впакування вміщуються в один безперервний буфер і виконується пересилка даних. Отримане таким чином повідомлення знову, тільки вже за допомогою методу розпакування даних, розподіляється по потрібних комірках пам'яті.

Для забезпечення такого способу пересилання даних, до стандарту MPI включені три процедури: `MPI_PACK`, `MPI_UNPACK` і `MPI_PACK_SIZE`.

Відповідно до класу `Comm` пакета MPJ включені три методи-аналоги цих процедур.

Перший з них:

```
public int Pack(Object inbuf, int offset, int incount,
                Datatype datatype, mpjbuf.Buffer outbuf,
                int position) throws MPIException
```

дозволяє упакувати `incount` елементів типа `datatype` з масиву `inbuf` у масив `outbuf`, починаючи з позиції `position` байт від початку цього масиву. Буфер `outbuf` повинен бути об'єктом спеціального типа `mpjbuf.Buffer` і мати розмір достатній щонайменше для розміщення всієї кількості `outsize` байтів з вихід-

ного буфера. Метод повертає значення, яке дорівнює новій позиції – `position` у буфері `outbuf`, що є доступною для подальшого упакування даних. Тобто значення, яке вертається з методу, є більшим відносно значення `position` переданого до методу на число байт у зробленому запису. Повідомлення, що відповідає методу, передається до об'єкта комунікатора, у якому надалі буде виконуватися операція комунікації з пересиланням даних. Для пересилання таких впакованих даних використовується наперед визначений тип `MPI.PACKED`.

Зворотна операція розпакування та розміщення даних процесом одержувачем повідомлення виконується методом з оголошенням:

```
public int Unpack(mpjbuf.Buffer inbuf, int position,
                  Object outbuf, int offset, int outcount,
                  Datatype datatype) throws MPIException
```

Метод розпаковує `outcount` елементів типа `datatype` з масиву `inbuf` типа `mpjbuf.Buffer` починаючи з позиції `position` байт від початку масиву в масив `outbuf` зі зміщенням `offset` у остатньому. Значення, яке вертається з методу відповідає значенню `position` зменшеному на кількість витягнутих байтів. Масив `inbuf` мусить мати такий розмір щоб бути спроможним вмістити всі дані, що підлягають розпаковці. Метод, як і попередній, також є членом об'єкта-комунікатора у якому виконувалася операція комунікації.

Третій зі згаданих вище аналогів – метод

```
public int Pack_size(int incount, Datatype datatype)
                                        throws MPIException
```

слугує для визначення обсягу `size` оперативної пам'яті (у байтах) необхідного для впакування `incount` елементів типа `datatype`. Обчислений цим методом розмір пам'яті, потрібній для впакування, може перевищувати суму розмірів елементів даних, що підлягають пакуванню.

Наступна програма (Ex_21) демонструє застосування розглянутих методів. Тут масив `buf` використовується як буфер для впакування методом `Pack`

arrLen (передається у програму як параметр командної строки) елементів масиву a типа double й arrLen елементів масиву b типа char. Отримане повідомлення пересилається звичайним методом Bcast від процесу 0 до всіх інших процесів. Останні, за допомогою звернення до методів Unpack, розпаковують отримане повідомлення.

```
import mpi.*;
import mpjbuf.*;
import java.util.*;
class Ex_21 {
  public static void main(String[] args) throws MPIException {
    int rank, position;
    String[] arg = MPI.Init(args);
    int arrLen = Integer.parseInt(arg[0]);
    double[] a = new double[arrLen];
    char[] b = new char[arrLen];
    int bufSize = MPI.COMM_WORLD.Pack_size(a.length, MPI.DOUBLE) +
                  MPI.COMM_WORLD.Pack_size(b.length, MPI.CHAR);
    Buffer buf = new Buffer(bufSize);
    rank = MPI.COMM_WORLD.Rank ();
    for(int i = 0; i < arrLen; i++) {
      a[i] = (double)rank + 1.0;
      if(rank == 0) b[i] = 'a'; else b[i] = 'b'; }
    position = 0;
    if(rank == 0) {
      position = MPI.COMM_WORLD.Pack(a, 0, a.length, MPI.DOUBLE,
                                     buf, position);
      position = MPI.COMM_WORLD.Pack(b, 0, b.length, MPI.CHAR,
                                     buf, position);
      MPI.COMM_WORLD.Bcast(buf, 0, bufSize, MPI.PACKED, 0); }
    else {
      MPI.COMM_WORLD.Bcast(buf, 0, bufSize, MPI.PACKED, 0);
      position = MPI.COMM_WORLD.Unpack(buf, position, a, 0,
                                       a.length, MPI.DOUBLE);
      position = MPI.COMM_WORLD.Unpack(buf, position, b, 0,
                                       b.length, MPI.CHAR); }
    System.out.printf("\nПроцес %d\n a[] = %s\n b[] = %s\n", rank,
                      Arrays.toString(a), Arrays.toString(b));
    MPI.Finalize();
  }
}
```

5.7.3 Контрольні запитання та завдання для самостійної роботи

– Які спеціальні способи пропонується стандартом MPI для виконання пересилання різнотипних даних?

– Розкажіть як і коли виконується створення похідних типів даних.

– На якому етапі виконання програми з'являється можливість використання похідні типи даних?

– Що таке узагальнений похідний тип даних? Розкажіть про нього докладніше.

– Що називається шаблоном і сигнатурою типа? Як можна собі уявити наперед визначені базові типи даних?

– Що таке екстент типа, як він визначається і як між собою співвідносяться екстент і розмір типа?

– Розкажіть про тип даних, який створюється методом Contiguous?

– Що з себе являє тип, створений методом Vector? Охарактеризуйте цей метод.

– Чим відрізняються типи Vector і Hvector?

– Коли може виникнути потреба в використанні методу Indexed? Докладно опишіть цей метод.

– Опишіть тип даних Struct, розкажіть навіщо він потрібний.

– Розкажіть про впакований тип даних, як він створюється і як з ним працювати.

– Створіть похідний тип даних для пересилань діагональної матриці.

– Напишіть вихідний код для пересилання від всіх процесів застосування нульовому процесу такої структури даних, яка складається з рангу процесу й назви вузла, на якому даний процес запущений (назва вузла, отримується за допомогою методу Get_processor_name).

– Прямокутна матриця розподілена по процесах по стовпцях. Спробуйте написати вихідний код для перестановки стовпців у зворотному порядку, використовуючи для пересилань похідний тип даних.

– Виконайте попереднє завдання з використанням пересилань упакованих даних.

– У чому, на ваш погляд, полягають переваги й недоліки використання пересилань упакованих даних у порівнянні з пересиланнями даних похідних типів?

– З використанням похідних типів даних напишіть програму транспонування матриці.

ЛІЦЕНЗІЯ НА ПАКЕТ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ MPJ EXPRESS ВЕРСІЇ 0.44

1. The license statement.

The MIT License

Copyright (c) 2005 - 2011

1. Distributed Systems Group, University of Portsmouth (2005)
2. Aamir Shafi (2005 - 2011)
3. Bryan Carpenter (2005 - 2011)
4. Mark Baker (2005 - 2011)

The bulk of code in this distribution was developed by the Distributed Systems. Group at the University of Portsmouth. Some sections of the code like the buffering API and derived datatypes include contributions developed at the Community Grids Lab at Indiana University.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

2. MPJ uses one third party software: the Java Service Wrapper project.

The license for the Java service wrapper project can be seen at $MPJ_HOME/THIRDPARTYLICENSES file.

ПЕРЕЛІК ПОСИЛАНЬ

1. Рольщиков В.Б. Технології розподілених систем та паралельних обчислень: Конспект лекцій (Змістовний модуль №2) – Одеса: ОДЕКУ, 2016. – 155 с.
2. Рольщиков В.Б. Технології розподілених систем та паралельних обчислень: Конспект лекцій (Змістовний модуль №1) – Одеса: ОДЕКУ (незатверджений).
3. Букатов А.А., Дацюк В.Н., Жегуло А.И. Программирование многопроцессорных вычислительных систем – Ростов-на-Дону: ООО “ЦВВР”, 2003. – 208 с.
4. Шпаковский Г.И., Серикова Н.В. Программирование для многопроцессорных систем в стандарте MPI: Учебное пособие – Минск: БГУ, 2002. – 323 с.
5. MPI: A Message-Passing Interface Standard Version 3.1 – Knoxville, Tennessee: Message Passing Interface Forum, June 4, 2015. – 836 p.
6. MPI: A Message-Passing Interface Standard [Електронний ресурс] – Режим доступу: http://www.mcs.anl.gov/research/projects/mpi/mpi-standard/mpi-report-1.1/mpi-report.htm
7. User Guides for Linux and Windows [Електронний ресурс] – Режим доступу: http://mpjexpress.org/guides.html
8. Воеводин В.В., Воеводин В.В. Параллельные вычисления – СПб.: БХВ-Петербург, 2002. – 317 с.
9. Корнеев В.Д. Параллельное программирование в MPI – Новосибирск: Изд-во СО РАН, 2000. – 184 с.
10. Parallel Compiler Runtime Consortium, HPCC and Java – a report by the Parallel Compiler Runtime Consortium, May 1996 [Електронний ресурс] – Режим доступу: http://www.npac.syr.edu/users/gcf/hpja va3.html
11. Лицензия MIT [Електронний ресурс] – Режим доступу: https://ru.wikipedia.org/wiki/ Лицензия_MIT
12. A.Shafi, J.Manzoor, K.Hameed, B.Carpenter, M.Baker Multicore-enabling the MPJ Express Messaging Library // PPPJ 2010: 8th International Conference on

the Principles and Practice of Programming in Java – Vienna, Austria: September 15-17 2010. – P 49-58

13. Антонов А.С. Введение в параллельные вычисления: Методическое пособие – М: НИВЦ МГУ им. М.В.Ломоносова, 2002. – 69 с.
14. Рольщиков В.Б. Застосування засобів середовища розробки Java при виконанні лабораторних робіт з об’єктно-орієнтованого програмування: Методичні вказівки – Одеса: ОДЕКУ, 2011. – 125 с.

АЛФАВІТНИЙ ПОКАЖЧИК ПО МЕТОДАХ, КОНСТАНТАХ ТА ДЕЯКИХ КЛАСАХ ПАКЕТА MPJ

A

Abort....71, 72
Allgather....61, 123, 140
Allgatherv....61, 123, 141
Allreduce....61, 123, 158
Alltoall....123, 142
Alltoallv....123, 143
ANY_SOURCE – константа...57, 67, 87
ANY_TAG – константа....67, 87
Attr_get.... 62

B

BAND – константа.... 145
Barrier....61, 87, 123, 125
Bcast....61, 124, 128
BOOLEAN – константа....60, 80
BOR – константа.... 145
Bsend....59, 83
Bsend_init....59, 112
BSEND_OVERHEAD – константа .. 84, 85
Buffer_attach....60, 84
Buffer_detach....60, 84
BXOR – константа.... 146
BYTE – константа....60, 80

C

Call.... 154
Cancel.... 59
CartParms – клас.... 182
CHAR – константа.... 60, 80
Clone.... 62
COMM_NULL – константа.... 171, 173, 178, 185
COMM_SELF – константа.... 171
COMM_WORLD – константа.... 59, 65, 70, 166, 171
Compare....61, 167
Contiguous.... 195
Coords....63, 180
Create.... 171
Create_cart....62, 177
Create_graph....63, 184
Create_intercomm.... 62

D

Difference....61, 165
Dims_create....63, 179
DOUBLE – константа.... 60, 81
DOUBLE2 – константа....81, 150

E

Excl....61, 164
Extent.... 201

F

Finalize....59, 69, 70, 90, 106

FLOAT – константа 60, 81
FLOAT2 – константа 81, 152
Free 62, 173
free (клас Datatype) 200
free (клас Group) 168
free (клас Op) 155

G

Gather 61, 124, 130, 132
Gatherv 61, 124, 132
Get 63
Get (клас CartParms) 182
Get (клас GraphParms) 187
Get_count.....88, 92
Get_processor_name 59, 74
GraphParms – клас 187
Group 163
GROUP_EMPTY – константа 62, 162, 163, 164
GROUP_NULL – константа 162

H

Hindexed..... 198
Hvector 197

I

Ibsend 59, 100
IDENT – константа 167
Incl.....61, 164
Indexed 197
Init 59, 69, 105
Initialized 59, 69, 70, 71
INT – константа 60, 81
INT2 – константа 81
Intersection.....61, 165
Iprobe.....59, 87, 101
Irecv 59, 100, 106, 117
Irsend 59, 100
Isend 59, 99
Issend 100

L

LAND – константа..... 146
Lb..... 200
LB – константа..... 60
LONG – константа..... 60, 81
LONG2 – константа..... 81
LOR – константа..... 146
LXOR – константа 146

M

Map (клас Cartcomm)..... 63
Map (клас Graphcomm) 63
MAX – константа 146
MAXLOC – константа.....146, 150
Merge..... 62
MIN – константа 146
MINLOC – константа 146, 150, 152

N

Neighbors.....63, 186
NULL_PROCESS – константа 60

O

OBJECT – константа 60, 81
Op – клас 154

P

Pack 60, 203
Pack_size 85, 204
PACKED – константа 60, 81
Probe.....59, 87, 93, 94
PROC_NULL – константа 183
PROD – константа 145

R

Range_excl 61
Range_incl 61
Rank.....59, 61, 63, 105
Rank (клас Cartcomm) 181
Rank (клас Comm) 72
Rank (клас Group)..... 167
Recv.....59, 87, 115
Recv_init.....112
Reduce 61, 123, 124, 144
Reduce_scatter 61, 158
Remote_group..... 62
Remote_size..... 62
Rsend 59, 86
Rsend_init.....59, 112

S

Scan 61, 124, 159
Scatter 61, 124, 134
Scatterv 61, 124, 134, 139
Send 57, 59, 79, 106
Send_init 59, 111
Sendrecv 59, 117
Sendrecv_replace 59, 118
Shift 63, 182
ShiftParms – клас 183
SHORT – константа 60, 80
SHORT2 – константа 81, 150
SIMILAR – константа..... 167
Size 59, 61, 105
Size (клас Comm)..... 72
Size (клас Datatype).....156, 200
Size (клас Group) 166
Split.....61, 172
Ssend..... 59, 86
Ssend_init 59, 112
Startall.....113
Status – клас 79
Struct..... 198
Sub.....63, 181
SUM – константа 145

T

Test 59, 62, 107
Test_Inter 62
Testall..... 107
Testany..... 107
Testsome 107
Topo_test.....63, 177
Translate_ranks 61, 167

U

Ub 201
UB – константа 60
UNDEFINED – константа 173
UNEQUAL – константа..... 167
Union 61, 165
Unpack.....60, 203
User_function – клас 153

V

Vector..... 196

W

Wait.....59, 102
Waitall.....59, 102
Waitany 59, 102, 104
Waitsome 59, 102, 104, 106
Wtick 59, 74
Wtime 59, 73

ПЕРЕЛІК РИСУНКІВ

Рисунок 1.1 − Структура каталогів пакета класів MPJ Express Version 0.44..... 13
Рисунок 1.2 − Зовнішній вигляд початкової сторінки інтерактивної допомоги пакету MPJ Express 14
Рисунок 1.3 − Комунікації з використанням сокетів на багатоядерній машині або машині з загальною пам'яттю [12] 15
Рисунок 1.4 − Можливі конфігурації запуску програм на основі MPJ Express .. 16
Рисунок 1.5 − Початок процедури встановлення значень змінних оточення ОС а) – вибір пункту «Панель управления» в основному меню системи; б) – зовнішній вигляд вікна «Панель управления» і вибір категорії «Производительность и обслуживание» 18
Рисунок 1.6 − Зовнішній вигляд вікон «Производительность и обслуживание» з вибором завдання «Просмотр информации об этом компьютере» – а) і «Свойства системы» з вибраною вкладкою «Дополнительно» – б) 18
Рисунок 1.7 − Загальний вигляд вікна «Переменные среды» з кнопкою «Создать» в блоці «Системные переменные» – а) і вікно «Новая системная переменная» з введеними ім'ям і значенням змінної MPJ_HOME – б) 19
Рисунок 1.8 − Демонстрація процесу змінення змінної PATH а) – вибір існуючого значення змінної у вікні «Переменные среды»; б) доповнення значення новим шляхом «;%MPJ_HOME%\bin;» 20
Рисунок 1.9 − Різні структури кластерів з якими може працювати MPJ Експрес а) – структура з розподіленою пам'яттю, включаючи кластери і мережі комп'ютерів; б) – гібридна конфігурація кластера з багатоядерних машин 21
Рисунок 2.1 − Початок створення нового Java-проекту 33
Рисунок 2.2 − Продовження створення Java-проекту "MulticoreDemo" 33
Рисунок 2.3 − Вкладка Libraries діалогового вікна Java Settings 34

Рисунок 2.4 − Вибір вікна конфігурування проекту 35
Рисунок 2.5 − Початковий вигляд вікна конфігурування проекту, вибір конфігурації запуску проекту 35
Рисунок 2.6 − Вигляд вікна конфігурування після вибору конфігурації запуску проекту 36
Рисунок 2.7 − Зовнішній вигляд вкладки Arguments 37
Рисунок 2.8 − Вигляд вікна вибору змінної оточення а – змінна оточення MPJ_HOME раніш вже встановлена; б – змінної MPJ_HOME у системі ще не існує 38
Рисунок 2.9 − Вікно створення і редагування змінних оточення 38
Рисунок 2.10 − Вікно встановлення параметрів нової змінної оточення 39
Рисунок 2.11 − Повністю введений аргумент віртуальної машини для роботи у багатоядерному режимі 39
Рисунок 2.12 − MPJ Express Debugger підключений до IDE Eclipse 44
Рисунок 2.13 − Відкриття вікна Run Configurations 44
Рисунок 2.14 − Встановлення параметрів для багатоядерній конфігурації 45
Рисунок 2.15 − Параметри запуску MPJ Express застосування у гомогенній кластерній конфігурації 46
Рисунок 2.16 − Ввід параметрів MPJ при локальному налагодженні 47
Рисунок 2.17 − Вигляд вікна перспективи Debug 48
Рисунок 2.18 − MPJ Parameters for Remote Debugging 49
Рисунок 2.19 − Параметри зв’язку для багатоядерного режиму віддаленого налагодження 50
Рисунок 2.20 − Параметри зв’язку для віддаленого налагодження у розподіленому режимі 51
Рисунок 3.1 − UML діаграма структури пакету MPJ Express 52
Рисунок 3.2 − UML діаграма пакету mpi 53
Рисунок 3.3 − Дерево спадкування класів в пакеті mpi 54
Рисунок 3.4 − Діаграма композиції для класу MPI пакету mpi 55
Рисунок 4.1 − Схема пересилання даних методом Bcast 123

Рисунок 4.2 – Ілюстрація роботи методу Bcast 123
Рисунок 4.3 – Схема збору даних методом Gather 125
Рисунок 4.4 – Ілюстрація виконання методу Gather..... 126
Рисунок 4.5 – Ілюстрація збирання даних методом Gatherv..... 128
Рисунок 4.6 – Схема розподілення даних методом Scatter 129
Рисунок 4.7 − Розподіл даних з великого буфера виділеного процесу рівними частками по маленьким буферам всіх процесів комунікатора... 129
Рисунок 4.8 – Ілюстрація дії метода Scatterv..... 133
Рисунок 4.9 – Схема дії методу Allgather..... 135
Рисунок 4.10 – Схема пересилання даних методом Alltoall..... 136
Рисунок 4.11 – Ілюстрація збору даних методом Reduce одночасно з виконанням деяких операцій над ними..... 138
Рисунок 4.12 – Можливе представлення графа п’ятьох процесів а) – зображення графа, б) – його формальний опис..... 177
Рисунок 4.13 − Схема розрахунків елементів масивів index і edges для передачі у метод Create_graph..... 178
Рисунок 4.14 − Структура комунікаційного буферу для типа даних, створеного за допомогою метода Contiguous 187

ПЕРЕЛІК ТАБЛИЦЬ

Таблиця 3.1 − Операції групи загального призначення 57
Таблиця 3.2 − Типи операцій з групи повідомлень «точка-точка» 57
Таблиця 3.3 − Група різних колективних операцій 58
Таблиця 3.4 − Операції з групами 58
Таблиця 3.5 − Операції віддаленої комунікації 59
Таблиця 3.6 − Дії з віртуальними топологіями процесів 59
Таблиця 4.1 − Типи даних у пакеті MPJ Express у порівнянні з примітивними і стандартними 76
Таблиця 4.2 − Визначені константи операції у методі `Reduce` 139
Таблиця 4.3 − Пояснення застосування методу `Dims_create` 172